

1 Introdução

Foram implementados e testados diferentes encoders espaciais da biblioteca *TorchSpatial*. A seguir estão os encoders considerados.

- Tile
- WRAP
- Space2Vec-grid
- Sphere2Vec-C
- Sphere2Vec-M
- SIREN

2 Estados Considerados

Para maior variação e por conta do tempo de execução desses algoritmos, foram escolhidos os seguintes Estados de pequeno e médio porte.

- Alaska
- Alabama
- Florida
- California

3 Configuração Experimental

Assumindo entradas de latitude/longitude (`lat`, `lon`) em graus, os modelos produzem um vetor $\mathbf{z} \in R^d$ que é utilizado como embedding espacial. Todos os modelos treinam com a mesma forma de aprendizado contrastivo e também com o mesmo modelo de geração de embeddings finais, com 64 dimensões.

```
[caption={Implementao de camadas Feed-Forward usadas aps todos os Encoders}]
import torch
import torch.nn as nn
from torch.nn import init
import torch.nn.functional as F

import torch.utils.data
import math

class LayerNorm(nn.Module):
    """
    Layer Normalization.
    Implementao simples de normalizao por camada,
    aplicada opcionalmente nos encoders espaciais.
    """
```

```

def __init__(self, feature_dim, eps=1e-6):
    super(LayerNorm, self).__init__()
    self.gamma = nn.Parameter(torch.ones((feature_dim,)))
    self.register_parameter("gamma", self.gamma)
    self.beta = nn.Parameter(torch.zeros((feature_dim,)))
    self.register_parameter("beta", self.beta)
    self.eps = eps

def forward(self, x):
    # x: [batch_size, embed_dim]
    mean = x.mean(-1, keepdim=True)
    std = x.std(-1, keepdim=True)
    return self.gamma * (x - mean) / (std + self.eps) + self.beta

def get_activation_function(activation, context_str):
    if activation == "leakyrelu":
        return nn.LeakyReLU(negative_slope=0.2)
    elif activation == "relu":
        return nn.ReLU()
    elif activation == "sigmoid":
        return nn.Sigmoid()
    elif activation == "tanh":
        return nn.Tanh()
    elif activation == "gelu":
        return nn.GELU()
    else:
        raise Exception("{} activation not recognized.".format(context_str))

class SingleFeedForwardNN(nn.Module):

    def __init__(self, input_dim,
                 output_dim,
                 dropout_rate=None,
                 activation="sigmoid",
                 use_layernormalize=False,
                 skip_connection=False,
                 context_str=''):
        super(SingleFeedForwardNN, self).__init__()
        self.input_dim = input_dim
        self.output_dim = output_dim

        self.dropout = nn.Dropout(p=dropout_rate) if dropout_rate else None
        self.act = get_activation_function(activation, context_str)

        self.layernorm = nn.LayerNorm(self.output_dim) if use_layernormalize else None

        if self.input_dim == self.output_dim:
            self.skip_connection = skip_connection
        else:

```

```

        self.skip_connection = False

    self.linear = nn.Linear(self.input_dim, self.output_dim)
    nn.init.xavier_uniform_(self.linear.weight)

def forward(self, input_tensor):
    assert input_tensor.size()[-1] == self.input_dim
    output = self.linear(input_tensor)
    output = self.act(output)

    if self.dropout is not None:
        output = self.dropout(output)

    if self.skip_connection:
        output = output + input_tensor

    if self.layernorm is not None:
        output = self.layernorm(output)

    return output

class MultiLayerFeedForwardNN(nn.Module):

    def __init__(self, input_dim,
                 output_dim,
                 num_hidden_layers=0,
                 dropout_rate=None,
                 hidden_dim=-1,
                 activation="sigmoid",
                 use_layernormalize=False,
                 skip_connection=False,
                 context_str=None):

        super(MultiLayerFeedForwardNN, self).__init__()
        self.layers = nn.ModuleList()

        if num_hidden_layers <= 0:
            self.layers.append(
                SingleFeedForwardNN(
                    input_dim=input_dim,
                    output_dim=output_dim,
                    dropout_rate=dropout_rate,
                    activation=activation,
                    use_layernormalize=False,
                    skip_connection=False,
                    context_str=context_str
                )
            )
        else:
            self.layers.append(
                SingleFeedForwardNN(
                    input_dim=input_dim,

```

```

        output_dim=hidden_dim,
        dropout_rate=dropout_rate,
        activation=activation,
        use_layernormalize=use_layernormalize,
        skip_connection=skip_connection,
        context_str=context_str
    )
)

for _ in range(num_hidden_layers - 1):
    self.layers.append(
        SingleFeedForwardNN(
            input_dim=hidden_dim,
            output_dim=hidden_dim,
            dropout_rate=dropout_rate,
            activation=activation,
            use_layernormalize=use_layernormalize,
            skip_connection=skip_connection,
            context_str=context_str
        )
    )

self.layers.append(
    SingleFeedForwardNN(
        input_dim=hidden_dim,
        output_dim=output_dim,
        dropout_rate=dropout_rate,
        activation=activation,
        use_layernormalize=False,
        skip_connection=False,
        context_str=context_str
    )
)

def forward(self, input_tensor):
    output = input_tensor
    for layer in self.layers:
        output = layer(output)
    return output

```

```

def get_adaptive_interval(extent, target_divisions=200):

    min_lon, max_lon, min_lat, max_lat = extent

    width = max_lon - min_lon
    height = max_lat - min_lat

    max_dim = max(width, height)

    interval = max_dim / target_divisions

    km_size = interval * 111

```

```
    return interval
```

4 Contexto Geral

De forma genérica, cada codificador espacial no TorchSpatial segue o esquema:

$$Enc(x) = NN(PE(x)), \quad (1)$$

onde $PE(x)$ captura os aspectos espaciais relevantes da localização x , e NN é uma rede treinável responsável por refinar essa codificação para produzir a representação final desejada.

5 Modelos

5.1 Tile-based location context (Tile) [5]

O Tile Encoding é o método mais simples e que discretiza o espaço geográfico em regiões fixas, chamadas de *tiles*. Cada tile recebe um identificador único, e a localização é representada pelo tile ao qual pertence. Essa abordagem transforma coordenadas contínuas em categorias discretas, permitindo ao modelo aprender diferenças regionais.

A extensão considerada é feita por meio de um mapeamento de latitudes e longitudes mínima e máxima em um Estado, garantindo que os tiles se apresentem somente no limite geográfico dos próprios Estados e evite resultados inesperados. E além disso, é feito 200 divisões pois no artigo original esse é o valor usado para os Estados Unidos, então uma função foi feita para que esse número se preserve mas de acordo com as proporções do estado, retornando o intervalo.

Define-se uma função $PE_{tile}(x)$ que retorna um vetor one-hot correspondente ao índice do tile. Esse vetor é então mapeado para um embedding contínuo por meio de uma camada linear treinável. O Tile Encoding, no entanto, sofre com perdas nas fronteiras entre tiles, pois mesmo locais próximos terão embeddings necessariamente diferentes caso estiverem na fronteira entre 2 tiles distintos.

```
class GridLookupSpatialRelationLocationEncoder(nn.Module):
    def __init__(self,
                 spa_embed_dim,
                 extent,
                 interval=0.1,
                 coord_dim=2,
                 device="cuda",
                 ffn_act="relu",
                 ffn_num_hidden_layers=1,
                 ffn_dropout_rate=0.5,
                 ffn_hidden_dim=256,
                 ffn_use_layernormalize=True,
                 ffn_skip_connection=True
                 ):
        super().__init__()
        self.device = device

        self.position_encoder = GridLookupSpatialRelationPositionEncoder(
            extent=extent,
```

```

        interval=interval,
        device=device
    )

    self.lookup_embed = nn.Embedding(
        num_embeddings=self.position_encoder.num_cells,
        embedding_dim=ffn_hidden_dim
    )

    self.ffn = MultiLayerFeedForwardNN(
        input_dim=ffn_hidden_dim,
        output_dim=spa_embed_dim,
        num_hidden_layers=ffn_num_hidden_layers,
        dropout_rate=ffn_dropout_rate,
        hidden_dim=ffn_hidden_dim,
        activation=ffn_act,
        use_layernormalize=ffn_use_layernormalize,
        skip_connection=ffn_skip_connection,
        context_str="GridLookupEncoder"
    )

def forward(self, coords):
    cell_ids = self.position_encoder(coords)

    embeds = self.lookup_embed(cell_ids)

```

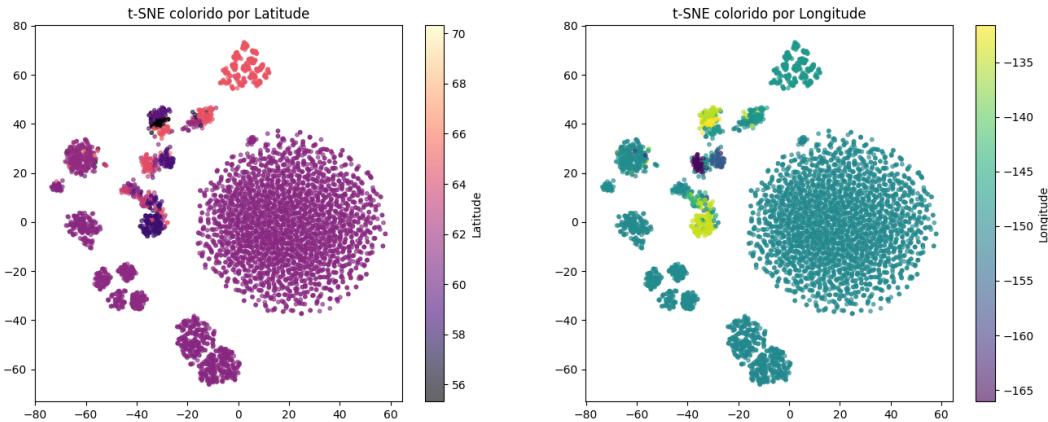


Figura 1: Visualização gerada do modelo Tile para Alaska.

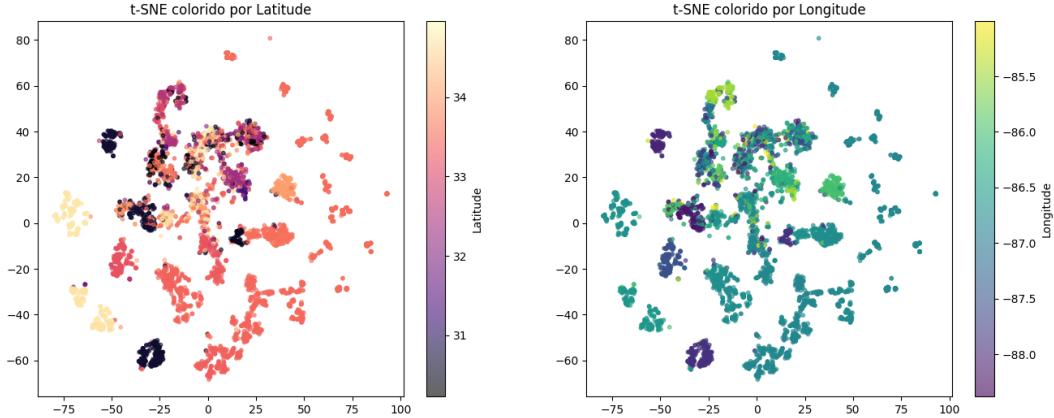


Figura 2: Visualização gerada do modelo Tile para Alabama

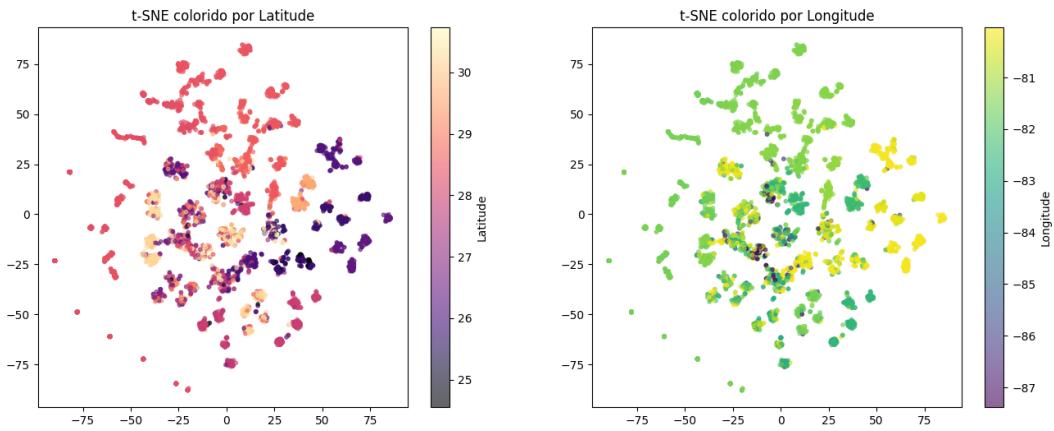


Figura 3: Visualização gerada do modelo Tile para Florida

5.2 WRAP: presence-only geographical priors (Wrap) [2]

Introduz uma codificação contínua e periódica das coordenadas geográficas, utilizando funções seno e cosseno com o objetivo de capturar a natureza cíclica da longitude e tratar adequadamente simetrias geográficas.

Dada uma localização $x = (\phi, \lambda)$, com latitude ϕ e longitude λ em radianos, define-se:

$$PE_{wrap}(x) = [\sin(\phi), \cos(\phi), \sin(\lambda), \cos(\lambda)]. \quad (2)$$

```
class WrapLocationEncoder(nn.Module):

    def __init__(self,
                 spa_embed_dim=64,
                 extent=None,
                 interval=None,
                 input_dim=4,
                 ffn_num_hidden_layers=1,
```

```

        ffn_dropout_rate=0.5,
        ffn_hidden_dim=256,
        ffn_act="relu",
        ffn_use_layernormalize=True,
        ffn_skip_connection=True,
        device="cuda"
    ):
        super().__init__()
        self.device = device

        self.position_encoder = WrapPositionEncoder(device=device)

        self.input_projector = nn.Linear(input_dim, ffn_hidden_dim)

        self.ffn = MultiLayerFeedForwardNN(
            input_dim=ffn_hidden_dim,
            output_dim=spa_embed_dim,
            num_hidden_layers=ffn_num_hidden_layers,
            dropout_rate=ffn_dropout_rate,
            hidden_dim=ffn_hidden_dim,
            activation=ffn_act,
            use_layernormalize=ffn_use_layernormalize,
            skip_connection=ffn_skip_connection,
            context_str="WrapEncoder"
        )

    def forward(self, coords):
        feat = self.position_encoder(coords)

        feat = self.input_projector(feat)

        out = self.ffn(feat)

```

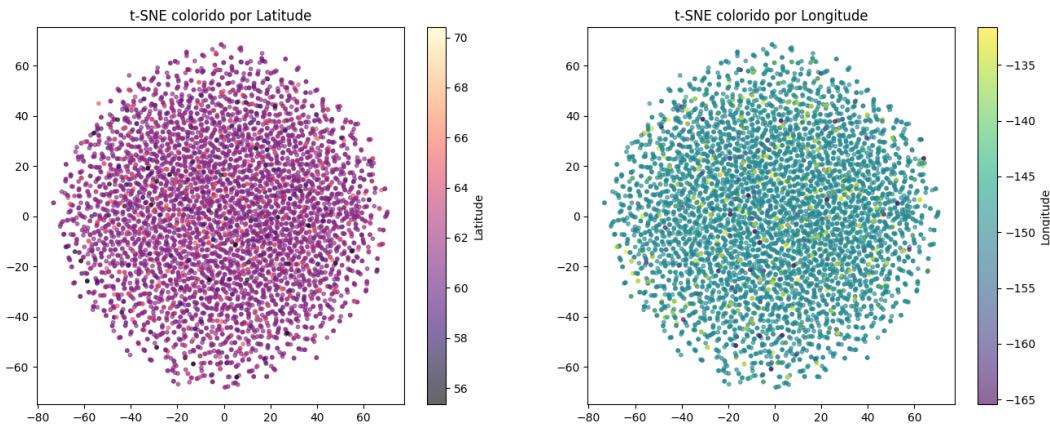


Figura 7: Visualização do modelo Wrap para Alaska.

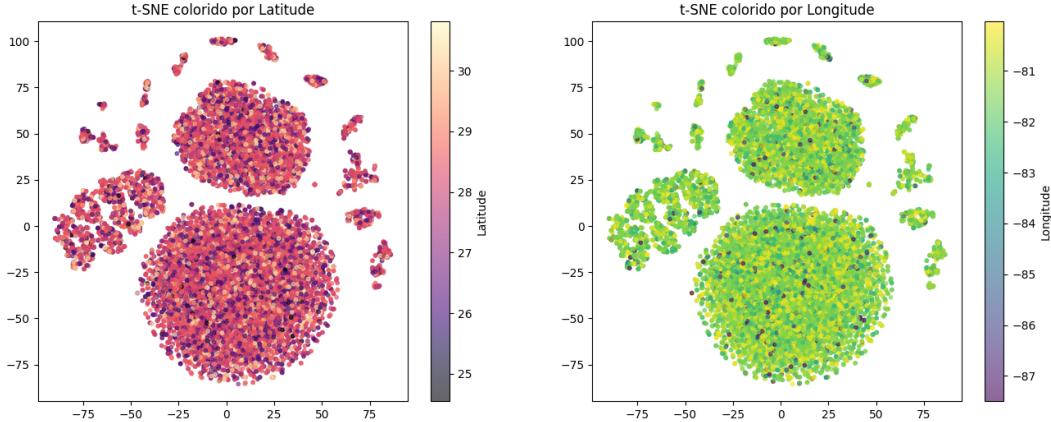


Figura 8: Visualização do modelo Wrap para Florida

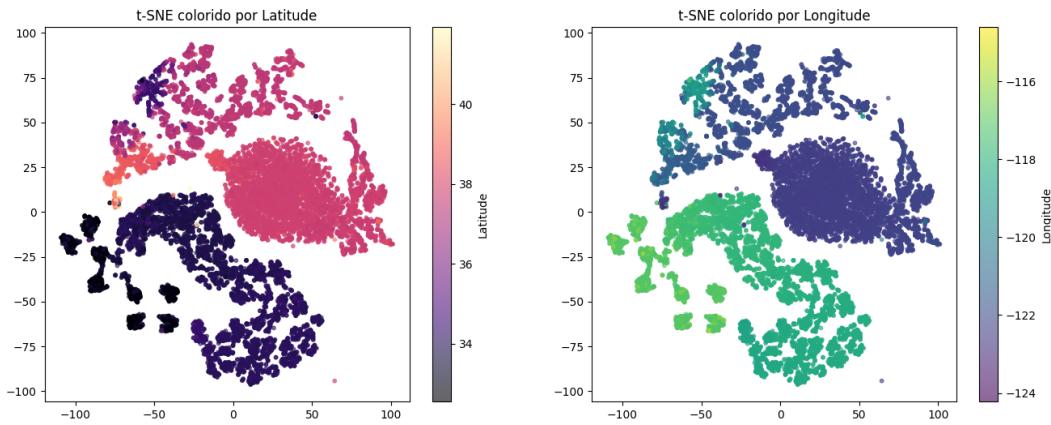


Figura 9: Visualização do modelo Wrap para California

5.3 Space2Vec-grid [4]

Usado originalmente e reposto para capturar relações espaciais em múltiplas escalas, inspirado em células de grid observadas em sistemas biológicos, utilizando múltiplas discretizações do espaço em diferentes resoluções, combinando codificações de várias escalas.

O Space2Vec, no entanto, não preserva distâncias geodésicas reais, por considerar um plano 2D e não uma esfera, o que gera distorções.

```
class GridCellSpatialRelationLocationEncoder(LocationEncoder):
    def __init__(
        self,
        spa_embed_dim=64,
        coord_dim=2,
        frequency_num=16,
        max_radius=10000,
        min_radius=10,
        freq_init="geometric",
        device="cuda",
        ffn_act="relu",
        ffn_num_hidden_layers=1,
```

```

        ffn_dropout_rate=0.5,
        ffn_hidden_dim=256,
        ffn_use_layernormalize=True,
        ffn_skip_connection=True,
        ffn_context_str="GridCellSpatialRelationEncoder",
    ):
        super().__init__(spa_embed_dim, coord_dim, device)
        self.frequency_num = frequency_num
        self.max_radius = max_radius
        self.min_radius = min_radius
        self.freq_init = freq_init
        self.ffn_act = ffn_act
        self.ffn_num_hidden_layers = ffn_num_hidden_layers
        self.ffn_dropout_rate = ffn_dropout_rate
        self.ffn_hidden_dim = ffn_hidden_dim
        self.ffn_use_layernormalize = ffn_use_layernormalize
        self.ffn_skip_connection = ffn_skip_connection

        self.position_encoder = GridCellSpatialRelationPositionEncoder(
            coord_dim=coord_dim,
            frequency_num=frequency_num,
            max_radius=max_radius,
            min_radius=min_radius,
            freq_init=freq_init,
            device=device,
        )
        self.ffn = MultiLayerFeedForwardNN(
            input_dim=self.position_encoder.pos_enc_output_dim,
            # input_dim=int(4 * frequency_num),
            output_dim=self.spa_embed_dim,
            num_hidden_layers=self.ffn_num_hidden_layers,
            dropout_rate=ffn_dropout_rate,
            hidden_dim=self.ffn_hidden_dim,
            activation=self.ffn_act,
            use_layernormalize=self.ffn_use_layernormalize,
            skip_connection=ffn_skip_connection,
            context_str=ffn_context_str,
        )

    def forward(self, coords):
        spr_embeds = self.position_encoder(coords)
        sprenc = self.ffn(spr_embeds)

        return sprenc

```

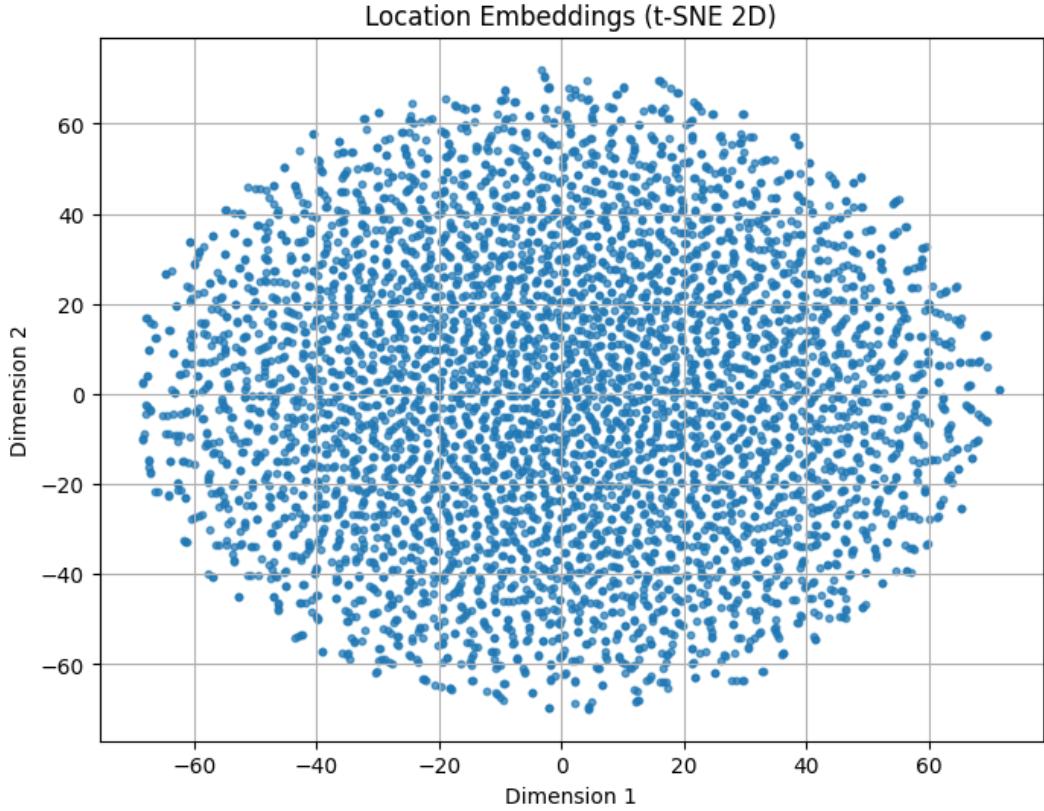


Figura 10: Visualização gerada do Space2Vec-grid para Alaska.

5.4 Sphere2Vec [3]

Estende o conceito de codificação multiescala para a superfície esférica da Terra, preservando distâncias geodésicas. A motivação é evitar distorções introduzidas por projeções planas, especialmente em aplicações globais.

Uma representação básica utiliza coordenadas cartesianas 3D na esfera unitária:

$$\mathbf{v}(x) = [\cos \phi \cos \lambda, \cos \phi \sin \lambda, \sin \phi]. \quad (3)$$

A distância Euclidiana entre esses vetores é uma função monotônica da distância de grande-círculo. O Sphere2Vec generaliza essa ideia usando múltiplas frequências e funções harmônicas esféricas, preservando a distância em múltiplas escalas, com maior ou menor definição. Neste caso, foi optado por 16 escalas diferentes, com o objetivo de captar melhor detalhes menores, especialmente envolvendo pequenas distâncias entre POIs a nível de bairro e rua.

```
class SphereLocationEncoder(nn.Module):

    def __init__(
        self,
        spa_embed_dim=64,

        extent=None,
        interval=None,
```

```

        num_scales=16,
        min_scale=1,
        max_scale=1000,
        ffn_num_hidden_layers=1,
        ffn_dropout_rate=0.5,
        ffn_hidden_dim=256,
        ffn_act="relu",
        ffn_use_layernormalize=True,
        ffn_skip_connection=True,
        device="cuda"
    ):

        super().__init__()
        self.device = device

        self.position_encoder = SpherePositionEncoder(
            min_scale=min_scale,
            max_scale=max_scale,
            num_scales=num_scales,
            device=device
        )

        input_dim = 3 * num_scales * 2

        self.input_projector = nn.Linear(input_dim, ffn_hidden_dim)

        self.ffn = MultiLayerFeedForwardNN(
            input_dim=ffn_hidden_dim,
            output_dim=spa_embed_dim,
            num_hidden_layers=ffn_num_hidden_layers,
            dropout_rate=ffn_dropout_rate,
            hidden_dim=ffn_hidden_dim,

```

5.4.1 Sphere2Vec-C [3]

Enfatiza componentes de *coordenadas puras*, por exemplo utilizando transformações do tipo $\sin \phi$, $\cos \phi$, $\sin \lambda$ e $\cos \lambda$ em múltiplas frequências radiais.

A codificação é baseada em uma projeção direta da posição geográfica (ϕ, λ) (lat, longitude) para coordenadas tridimensionais na esfera unitária:

$$(x, y, z) = (\cos \phi \cdot \cos \lambda, \cos \phi \cdot \sin \lambda, \sin \phi)$$

Essas coordenadas são então associadas a uma célula geodésica fixa através de uma malha esférica. Cada célula tem um vetor de embedding aprendido.

Esse processo introduz descontinuidades, posições próximas podem ser mapeadas para células diferentes, o que limita a suavidade da representação espacial.

```

class SpherePositionEncoder(nn.Module):

    def __init__(self, min_scale=1, max_scale=1000, num_scales=16, device='cuda'):
        super().__init__()
        self.device = device

```

```

self.num_scales = num_scales

scales = torch.logspace(np.log10(min_scale), np.log10(max_scale), num_scales)
self.register_buffer('scales', scales)

def forward(self, coords):
    if not torch.is_tensor(coords):
        coords = torch.tensor(coords, dtype=torch.float32).to(self.device)
    else:
        coords = coords.float().to(self.device)

    lat = coords[..., 0]
    lon = coords[..., 1]

    lat_rad = torch.deg2rad(lat)
    lon_rad = torch.deg2rad(lon)

    x = torch.cos(lat_rad) * torch.cos(lon_rad)
    y = torch.cos(lat_rad) * torch.sin(lon_rad)
    z = torch.sin(lat_rad)

    vec_3d = torch.stack([x, y, z], dim=-1)

    scaled_vecs = vec_3d.unsqueeze(-1) * self.scales

    sin_feat = torch.sin(scaled_vecs)
    cos_feat = torch.cos(scaled_vecs)

    emb = torch.cat([sin_feat, cos_feat], dim=-1).flatten(1)

    return emb

```

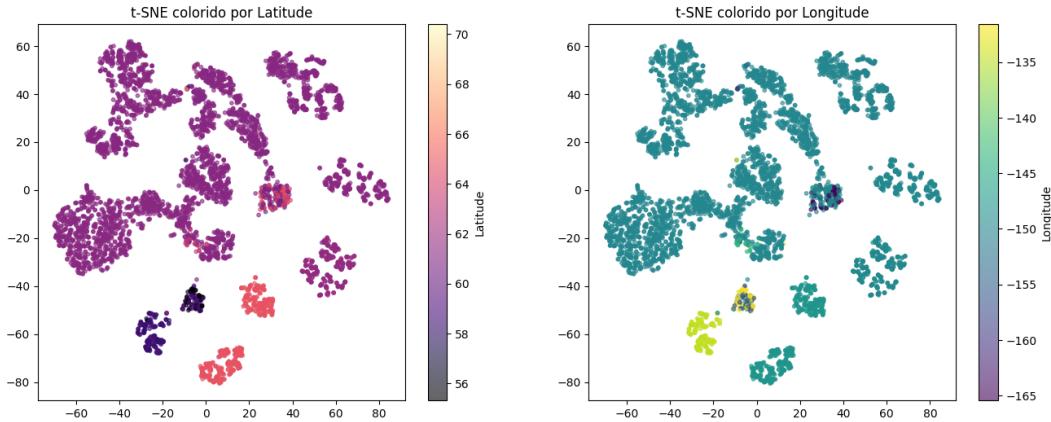


Figura 11: Visualização gerada do Sphere2Vec-C para Alaska.

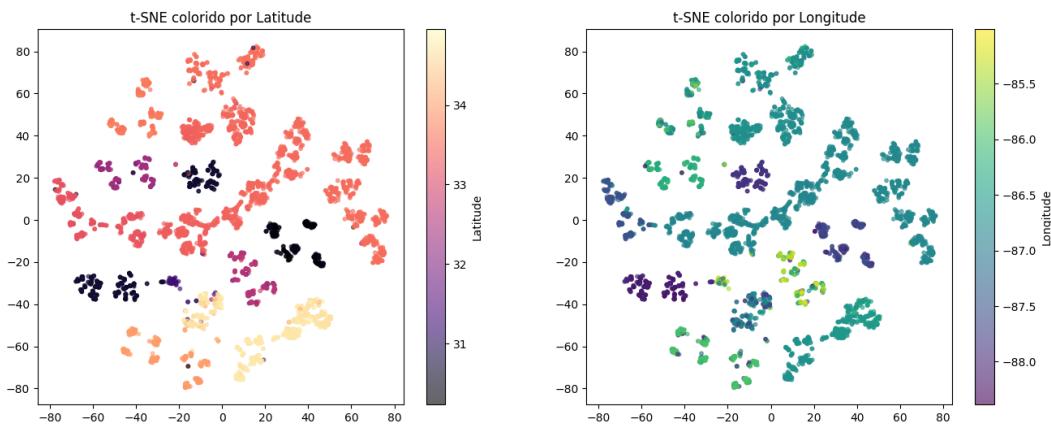


Figura 12: Visualização gerada do Sphere2Vec-C para Alabama.

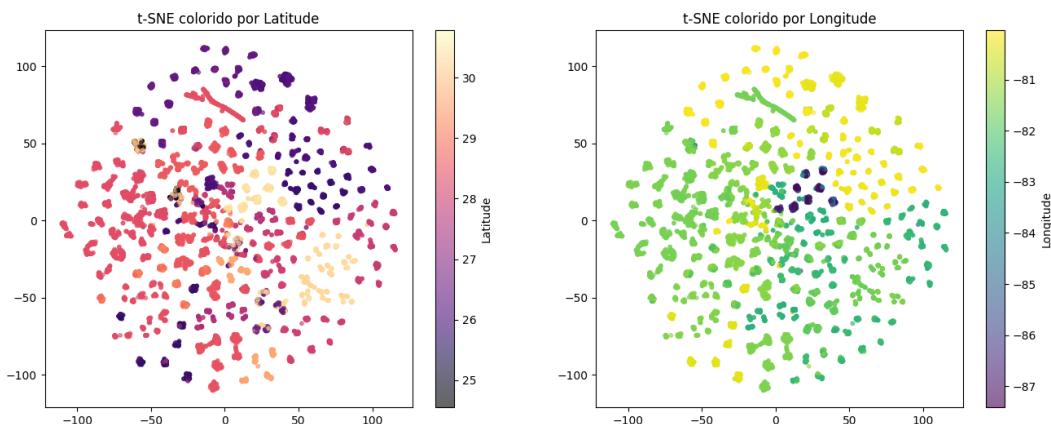


Figura 13: Visualização gerada do Sphere2Vec-C para Florida

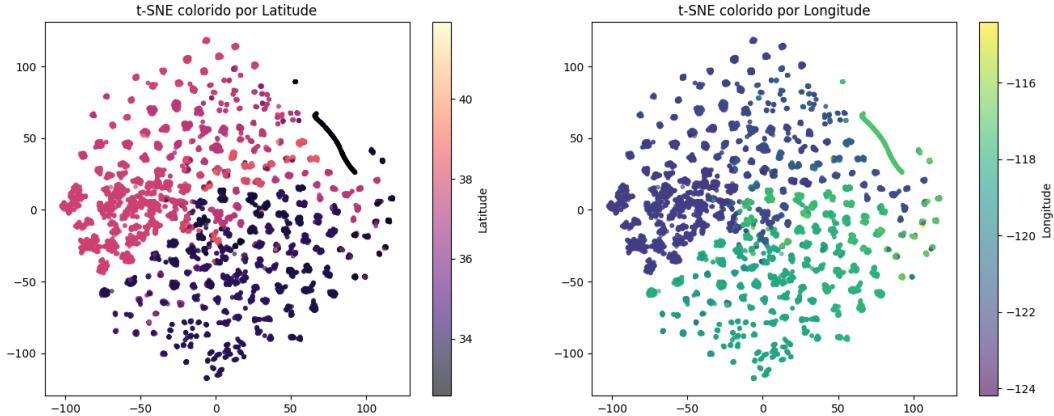


Figura 14: Visualização gerada do Sphere2Vec-C para California

5.4.2 Sphere2Vec-M [3]

Aplica encoding sobre superfície esférica, com parametrizações e ajustes internos. A localização geográfica é transformada diretamente em uma representação latente usando uma MLP. Para isso, as coordenadas (ϕ, λ) são convertidas em coordenadas cartesianas tridimensionais (x, y, z) na esfera unitária:

$$(x, y, z) = (\cos \phi \cdot \cos \lambda, \cos \phi \cdot \sin \lambda, \sin \phi)$$

Esses valores são usados como entrada de uma MLP, que aprende a gerar vetores embedding diretamente a partir da posição. Ao contrário da variante baseada em células (Sphere2Vec-C), essa abordagem evita discretização e oferece uma codificação suave e diferenciável do espaço geográfico.

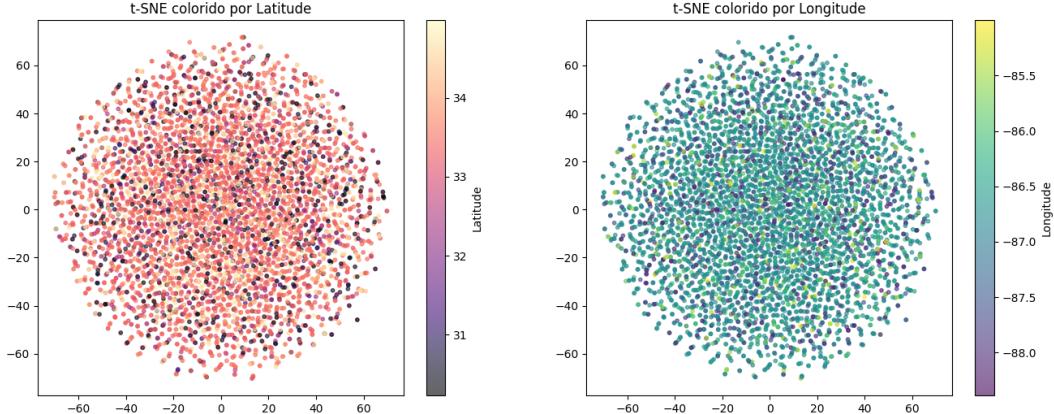


Figura 15: Visualização gerada do Sphere2Vec-M para Alabama.

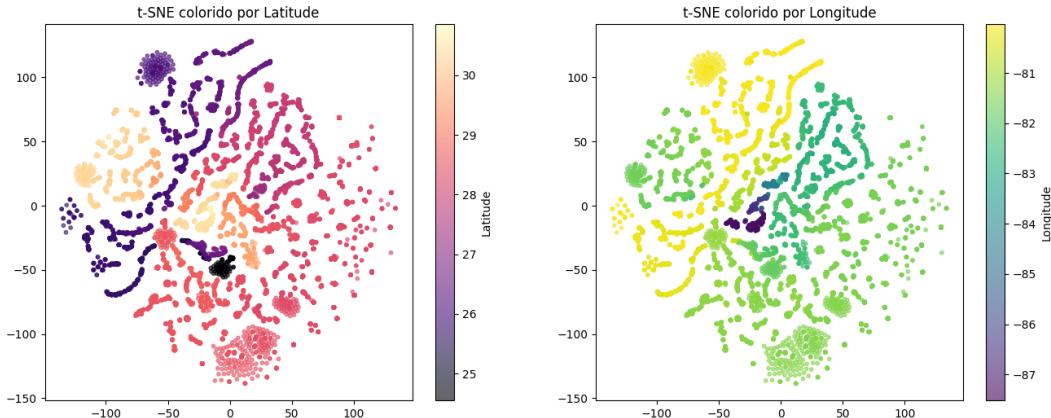


Figura 16: Visualização gerada do Sphere2Vec-M para Florida

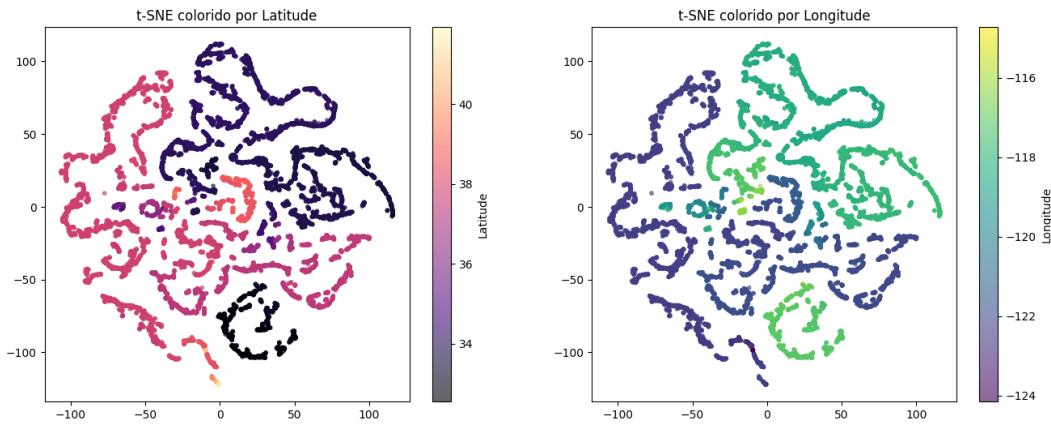


Figura 17: Visualização gerada do Sphere2Vec-M para California

```
class SpherePositionEncoder(nn.Module):

    def __init__(self, min_scale=1, max_scale=1000, num_scales=16, num_centroids=128,
                 device='cuda'):
        super().__init__()
        self.device = device
        self.num_scales = num_scales
        self.num_centroids = num_centroids

        scales = torch.logspace(np.log10(min_scale), np.log10(max_scale), num_scales)
        self.register_buffer('scales', scales)

        centroids = torch.randn(num_centroids, 3)
        centroids = torch.nn.functional.normalize(centroids, dim=-1)
        self.register_buffer('centroids', centroids)

    def forward(self, coords):
        if not torch.is_tensor(coords):
            coords = torch.tensor(coords, dtype=torch.float32).to(self.device)
        else:
```

```

        coords = coords.float().to(self.device)

        lat = coords[:, 0]
        lon = coords[:, 1]
        lat_rad = torch.deg2rad(lat)
        lon_rad = torch.deg2rad(lon)

        x = torch.cos(lat_rad) * torch.cos(lon_rad)
        y = torch.cos(lat_rad) * torch.sin(lon_rad)
        z = torch.sin(lat_rad)

        input_vec = torch.stack([x, y, z], dim=-1)

        dot_product = torch.matmul(input_vec, self.centroids.t())

        distance = 1.0 - dot_product
        weighted_dist = distance.unsqueeze(-1) * self.scales.view(1, 1, -1)

        rbf_feat = torch.exp(-weighted_dist)

        emb = rbf_feat.flatten(1)

    return emb

```

5.5 SIREN (SH) [1]

O modelo SIREN utiliza funções com redes neurais para representar e simplificar funções altamente oscilatórias de forma suave e contínua, combinando harmônicos esféricos como entrada com uma rede SIREN para gerar embeddings espaciais.

Diferentemente das abordagens como Tile ou WRAP, o SIREN não depende de discretizações rígidas do espaço ou suposições periódicas, é adaptativo e não depende de nenhuma restrição.

Um neurônio SIREN calcula:

$$y = \sin(Wx + b). \quad (4)$$

```

class SineLayer(nn.Module):

    def __init__(self, in_features, out_features, bias=True, is_first=False, omega_0=30):
        super().__init__()
        self.omega_0 = omega_0
        self.is_first = is_first

        self.linear = nn.Linear(in_features, out_features, bias=bias)

        self.init_weights()

    def init_weights(self):
        with torch.no_grad():
            if self.is_first:
                bound = 1 / self.linear.weight.size(1)
                self.linear.weight.uniform_(-bound, bound)

```

```

        else:

            bound = np.sqrt(6 / self.linear.weight.size(1)) / self.omega_0
            self.linear.weight.uniform_(-bound, bound)

    def forward(self, input):

        return torch.sin(self.omega_0 * self.linear(input))

class SirenLocationEncoder(nn.Module):
    def __init__(self,
                 spa_embed_dim=64,
                 ffn_hidden_dim=256,
                 ffn_num_hidden_layers=3,
                 omega_0=30,
                 device="cuda",
                 extent=None, interval=None, min_scale=None, max_scale=None
                 ):
        super().__init__()
        self.device = device

        layers = []

        layers.append(SineLayer(2, ffn_hidden_dim, is_first=True, omega_0=omega_0))

        for _ in range(ffn_num_hidden_layers):
            layers.append(SineLayer(ffn_hidden_dim, ffn_hidden_dim, is_first=False,
                                  omega_0=omega_0))

        self.net = nn.Sequential(*layers)

        self.final_projector = nn.Linear(ffn_hidden_dim, spa_embed_dim)

    def forward(self, coords):
        if not torch.is_tensor(coords):
            coords = torch.tensor(coords, dtype=torch.float32).to(self.device)
        else:
            coords = coords.float().to(self.device)

        x = coords / 180.0

        x = self.net(x)

```

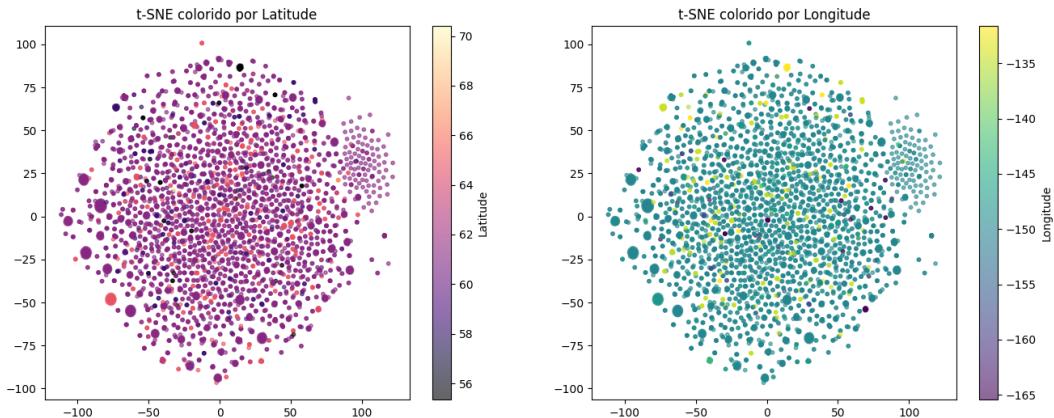


Figura 18: Visualização gerada do SIREN para Alaska.

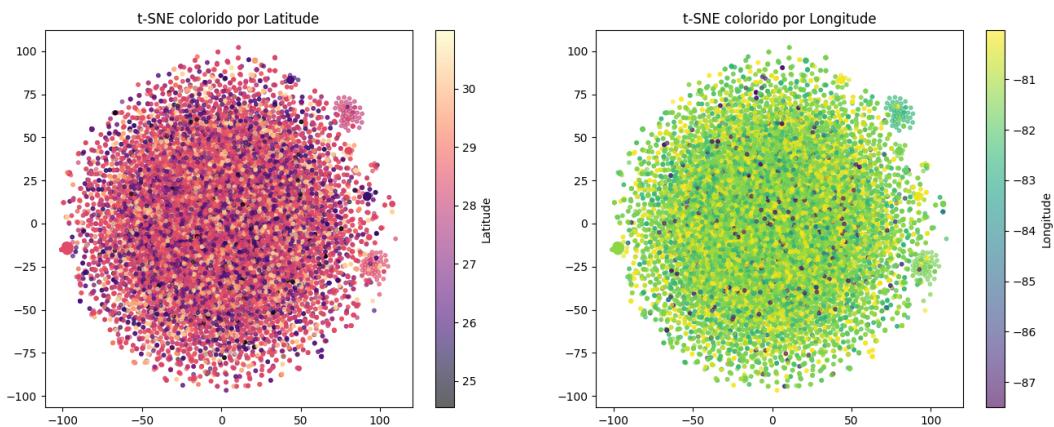


Figura 19: Visualização gerada do SIREN para Florida.

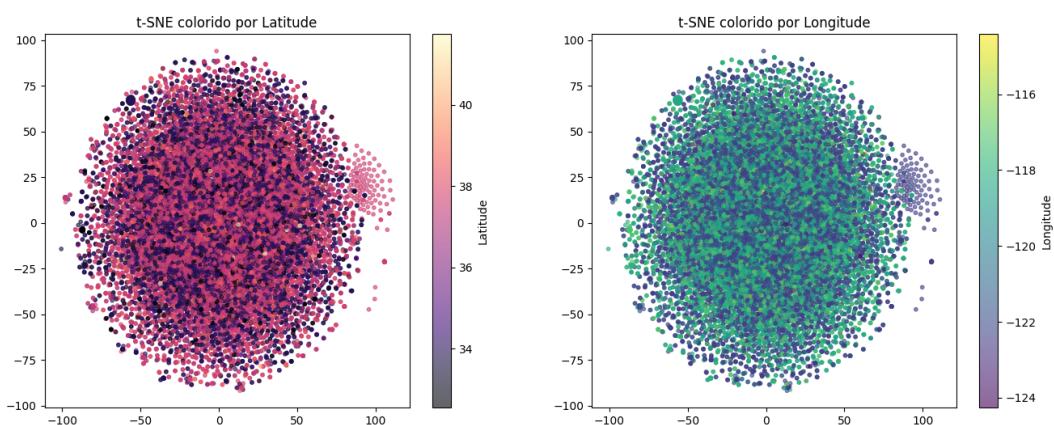


Figura 20: Visualização gerada do SIREN para California .

6 Resultados

6.1 Florida

Tabela 1: Tabela Comparando Resultados de Predição de Categoria (POI Category Prediction) para Florida

		TILE	WRAP	Sphere2Vec-C	Sphere2Vec-M	SIREN	Space2Vec
8*F1	Community	23.18 ± 0.74	15.73 ± 1.01	19.99 ± 0.11	33.80 ± 1.17	84.30 ± 1.08	65.11 ± 0.74
	Entertainment	28.53 ± 1.04	0.00 ± 0.00	19.94 ± 0.91	38.48 ± 1.52	90.68 ± 0.22	62.38 ± 0.79
	Food	43.49 ± 0.87	43.76 ± 0.57	43.32 ± 0.94	44.84 ± 1.15	83.87 ± 0.35	54.66 ± 0.48
	Nightlife	20.09 ± 0.93	11.23 ± 1.04	19.90 ± 0.55	23.42 ± 0.48	87.71 ± 0.63	41.76 ± 0.63
	Outdoors	0.00 ± 0.00	0.00 ± 0.00	0.05 ± 0.09	13.22 ± 0.35	83.31 ± 0.32	47.18 ± 1.54
	Shopping	38.39 ± 0.98	31.26 ± 0.75	35.86 ± 1.09	42.73 ± 1.46	80.24 ± 0.72	56.92 ± 0.41
	Travel	37.95 ± 0.46	3.54 ± 0.45	37.04 ± 0.73	50.35 ± 0.44	90.89 ± 0.24	70.23 ± 0.94
	macro avg	27.38 ± 0.20	15.07 ± 0.06	25.16 ± 0.20	35.27 ± 0.06	85.86 ± 0.38	56.03 ± -

Tabela 2: Tabela Comparando Resultados de Predição de Próxima Categoria (Next-POI Prediction) para Florida

		TILE	WRAP	Sphere2Vec-C	Sphere2Vec-M	SIREN	Space2Vec
8*F1	Community	40.05 ± 0.15	39.59 ± 1.52	38.90 ± 1.17	39.97 ± 1.08	39.63 ± 0.75	39.80 ± 0.96
	Entertainment	33.21 ± 0.64	33.07 ± 0.49	33.71 ± 0.37	33.26 ± 0.78	33.59 ± 0.83	33.01 ± 1.22
	Food	41.25 ± 1.91	39.82 ± 2.27	38.63 ± 1.51	41.23 ± 0.89	40.00 ± 1.84	40.39 ± 1.94
	Nightlife	22.62 ± 2.10	22.47 ± 3.14	21.16 ± 2.06	21.91 ± 2.57	23.29 ± 2.87	21.45 ± 1.21
	Outdoors	17.24 ± 2.50	18.64 ± 2.73	19.28 ± 3.42	18.44 ± 3.70	17.35 ± 0.64	20.33 ± 0.85
	Shopping	42.19 ± 2.24	44.12 ± 1.75	45.15 ± 0.37	42.89 ± 2.54	44.10 ± 2.23	44.31 ± 1.26
	Travel	44.75 ± 0.75	44.92 ± 0.75	45.36 ± 0.04	45.17 ± 0.75	45.14 ± 0.69	45.10 ± 1.49
	macro avg	34.47 ± 0.27	34.66 ± 0.23	34.60 ± 0.22	34.70 ± 0.30	34.73 ± 0.22	34.91 ± -

6.2 Alabama

Tabela 3: Tabela Comparando Resultados de Predição de Categoria (POI Category Prediction) para Alabama

		TILE	WRAP	Sphere2Vec-C	Sphere2Vec-M	SIREN	Space2Vec
8*F1	Community	41.22 ± 2.57	31.41 ± 1.32	39.07 ± 1.49	31.83 ± 0.93	75.90 ± 1.46	56.87 ± -
	Entertainment	13.55 ± 0.78	0.00 ± 0.00	3.71 ± 0.84	0.00 ± 0.00	66.57 ± 6.38	35.74 ± -
	Food	51.66 ± 0.28	50.40 ± 1.50	51.14 ± 0.67	51.22 ± 0.52	72.01 ± 1.76	56.84 ± -
	Nightlife	30.17 ± 5.96	14.68 ± 4.22	26.07 ± 4.16	11.50 ± 4.07	75.64 ± 3.50	33.97 ± -
	Outdoors	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	47.33 ± 10.79	16.66 ± -
	Shopping	34.35 ± 2.47	23.99 ± 5.20	31.17 ± 1.32	22.73 ± 1.85	62.54 ± 4.16	43.52 ± -
	Travel	33.22 ± 0.59	0.00 ± 0.00	21.14 ± 1.79	0.00 ± 0.00	67.34 ± 3.09	45.71 ± -
	macro avg	29.17 ± 0.89	17.21 ± 0.18	24.61 ± 0.30	16.75 ± 1.02	66.76 ± 3.75	41.33 ± -

Tabela 4: Tabela Comparando Resultados de Predição do Próximo POI (Next-POI Prediction) para Alabama

		TILE	WRAP	Sphere2Vec-C	Sphere2Vec-M	SIREN	Space2Vec
8*F1	Community	43.98 ± 2.44	44.96 ± 0.54	42.92 ± 1.52	42.35 ± 0.48	44.67 ± 1.39	45.13 ± –
	Entertainment	18.62 ± 2.14	19.62 ± 1.10	19.36 ± 3.80	18.72 ± 1.92	17.16 ± 3.32	18.83 ± –
	Food	42.95 ± 2.35	43.80 ± 2.63	45.30 ± 5.59	46.79 ± 2.41	46.69 ± 2.20	46.87 ± –
	Nightlife	19.50 ± 4.50	18.64 ± 1.42	18.47 ± 1.82	18.06 ± 1.85	19.99 ± 3.48	20.89 ± –
	Outdoors	13.18 ± 3.65	13.99 ± 4.81	16.88 ± 4.19	16.06 ± 3.64	13.55 ± 5.17	13.84 ± –
	Shopping	37.71 ± 4.98	40.73 ± 4.37	39.07 ± 1.02	40.13 ± 3.32	38.85 ± 1.88	38.20 ± –
	Travel	22.10 ± 3.56	21.09 ± 4.41	20.97 ± 2.34	18.56 ± 3.09	21.72 ± 2.32	23.28 ± –
	macro avg	28.29 ± 1.10	28.98 ± 0.31	29.00 ± 0.77	28.67 ± 0.88	28.95 ± 0.77	29.58 ± –

6.3 California

Tabela 5: Tabela Comparando Resultados de Predição de Categoria (POI Category Prediction) para California

		TILE	WRAP	Sphere2Vec-C	Sphere2Vec-M	SIREN	Space2Vec
8*F1	Community	30.35 ± 0.16	25.80 ± 1.53	27.32 ± 0.59	35.55 ± 0.07	82.46 ± 0.20	59.84 ± 0.65
	Entertainment	11.11 ± 1.53	0.00 ± 0.00	0.25 ± 0.17	19.48 ± 1.00	85.21 ± 0.43	58.97 ± 0.22
	Food	51.46 ± 0.12	50.68 ± 0.70	50.26 ± 0.97	52.67 ± 0.24	82.10 ± 0.07	60.84 ± 0.44
	Nightlife	21.86 ± 1.33	17.92 ± 0.39	20.08 ± 0.88	21.22 ± 0.30	87.20 ± 0.55	36.64 ± 0.42
	Outdoors	12.87 ± 0.42	4.26 ± 0.74	5.80 ± 1.40	22.57 ± 0.73	81.35 ± 0.76	57.41 ± 1.03
	Shopping	31.66 ± 0.31	23.73 ± 0.73	27.38 ± 3.02	32.53 ± 0.12	74.13 ± 0.28	49.06 ± 0.44
	Travel	44.23 ± 0.44	28.99 ± 0.62	26.64 ± 0.92	49.09 ± 0.47	90.21 ± 0.27	68.69 ± 0.64
	macro avg	29.08 ± 0.15	21.63 ± 0.09	22.53 ± 0.22	33.30 ± 0.05	83.24 ± 0.07	56.78 ± –

Tabela 6: Tabela Comparando Resultados de Predição de Próxima Categoria (Next POI Category Prediction) para California

		TILE	WRAP	Sphere2Vec-C	Sphere2Vec-M	SIREN	Space2Vec
8*F1	Community	33.25 ± 0.75	33.21 ± 1.05	33.46 ± 0.72	32.68 ± 0.86	32.85 ± 0.70	33.64 ± 1.47
	Entertainment	17.07 ± 1.21	17.65 ± 1.18	18.48 ± 0.51	17.83 ± 0.90	16.36 ± 1.21	16.13 ± 2.07
	Food	47.01 ± 0.28	47.84 ± 1.24	47.54 ± 0.96	47.67 ± 0.39	47.39 ± 0.58	47.23 ± 0.78
	Nightlife	14.61 ± 1.46	13.79 ± 3.21	14.68 ± 1.51	15.10 ± 0.85	15.10 ± 0.85	15.08 ± 0.88
	Outdoors	21.47 ± 0.48	20.78 ± 0.97	20.90 ± 1.76	21.40 ± 0.98	21.10 ± 0.44	21.88 ± 0.68
	Shopping	36.69 ± 1.43	35.69 ± 1.53	35.07 ± 1.80	36.36 ± 0.80	35.40 ± 0.77	37.53 ± 1.62
	Travel	36.30 ± 1.42	36.73 ± 1.15	36.90 ± 0.57	37.61 ± 0.21	36.20 ± 1.27	37.55 ± 0.82
	macro avg	29.49 ± 0.14	29.38 ± 0.24	29.57 ± 0.21	29.51 ± 0.02	29.20 ± 0.06	29.86 ± –

Referências

- [1] Geographic location encoding with spherical harmonics and sinusoidal representation networks. *arXiv preprint arXiv:2310.06743*, 2023.
- [2] Oisin Mac Aodha, Elijah Cole, and Pietro Perona. Presence-only geographical priors for fine-grained image classification. *arXiv preprint arXiv:1906.05272*, 2019.
- [3] Gengchen Mai et al. A general-purpose location representation learning over a spherical surface for large-scale geospatial predictions. *arXiv preprint arXiv:2306.17624*, 2023.
- [4] Gengchen Mai, Krzysztof Janowicz, Bo Yan, Rui Zhu, Blake Regalia, Mia Shi, Guoliang Cao, and Stefano Ermon. Multi-scale representation learning for spatial feature distributions using grid cells. *arXiv preprint arXiv:1803.08102*, 2018.

- [5] Kevin Tang, Manohar Paluri, Li Fei-Fei, Rob Fergus, and Lubomir Bourdev. Improving image classification with location context. *arXiv preprint arXiv:1505.03873*, 2015.