H.IAAC – HUB DE INTELIGÊNCIA ARTIFICIAL E ARQUITETURAS COGNITIVAS

SEGUNDO TERMO ADITIVO AO ACORDO DE PARCERIA n°

0200-09/2021/Softex/Eldorado/Unicamp/Arq.Cognitiva/PPI

META 4 – REPRESENTAÇÃO DE CONHECIMENTO

M 4.1 RELATÓRIO TÉCNICO COM IMPLEMENTAÇÃO DE PROVAS DE CONCEITO

CAMPINAS – SP

12 DE DEZEMBRO DE 2022

Sumário

1. Introdução

Este relatório apresenta as atividades que foram realizadas na etapa 1 da fase 2 da Meta 4, denominada Representação do Conhecimento, do projeto intitulado Arquiteturas Cognitivas.

Nas etapas anteriores, foi realizado um aprofundamento teórico em redução de dimensionalidade usando algoritmos manifold learning, e foi implementada e avaliada uma interface em Android que permita a integração entre redução de dimensionalidade, treinamento federado e as arquiteturas cognitivas. Nesta fase são apresentados os resultados de aprofundamento teórico e prático sobre os novos algoritmos de *deep manifold learning*, especificamente os autoencoders.

Para esta finalidade, foram definidos os seguintes objetivos específicos:

- Realizar uma revisão da literatura sobre os algoritmos deep manifold learning
- Levantamento de artigos de Deep Manifold Learning
- Realizar uma revisão da literatura sobre Autoencoders
- Implementar os algoritmos usando os datasets mais usados na literatura
- Estudar e adaptar a implementação dos algoritmos para reconhecimento de atividades humanas (HAR)
- Avaliar os resultados dos diferentes algoritmos autoencoders

1.1 Contribuições

Este trabalho contribui com o estudo e a implementação dos algoritmos autoencoders para a área de representação de conhecimento especificamente para atividades de reconhecimento de atividades HAR.

1.1 Organização do estudo

Inicialmente é apresentada uma revisão da literatura sobre representação do conhecimento de algoritmos deep manifold learning , e detecção de atividades humanas, seguido da metodologia e arquitetura utilizada para o desenvolvimento e os resultados obtidos. O código utilizado neste projeto pode ser encontrado no anexo A.

2. Revisão da literatura

Este capítulo apresenta uma revisão dos conceitos relevantes para o desenvolvimento deste trabalho. Inicialmente se faz uma introdução dos algoritmos que permitem a representação do conhecimento especificamente os Autoencoders.

2.1 Manifold Learning

Como foi explicado nos relatórios anteriores da meta 4, o aprendizado múltiplo está relacionado às técnicas algorítmicas de redução de dimensionalidade e pode ser dividido em métodos lineares e não lineares. Os métodos lineares incluem a análise de componentes principais (PCA) e dimensionamento multidimensional (MDS). A aprendizagem múltipla não linear inclui Isomap, tsne, umap entre outros. O processo algorítmico da maioria dessas técnicas consiste em três etapas: uma busca no vizinho mais próximo, uma definição de distâncias ou afinidades entre pontos (Alan 2012).

Umap (Healy and Melville 2020) surgiu para resolver algumas desvantagens dos algoritmos até então existentes como, por exemplo, não preservar a estrutura global dos dados e serem difíceis de escalar para dimensões muito grandes de variáveis e observações. No link se apresentam os fundamentos matemáticos e estatísticos do algoritmo umap.

2.2 Deep Manifold Learning

A seguir são listadas as principais arquiteturas que fazem parte do Deep Manifold Learning:

- Topological Autoencoders: a ideia é preservar a estrutura topológica usando redes Neurais e o esquema de Codificação-Decodificação.

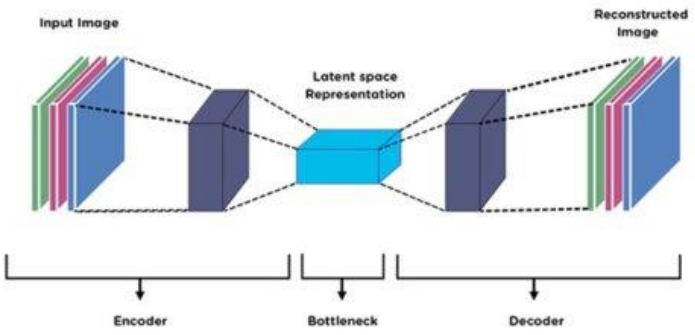


Figura 1: Arquitetura topological autoencoders

➢ Deep Isometric Manifold Learning: combina eficiência de Sparse MDS com a capacidade de generalizar de redes neurais Rede siamesa. (Preservação de geodésicas).

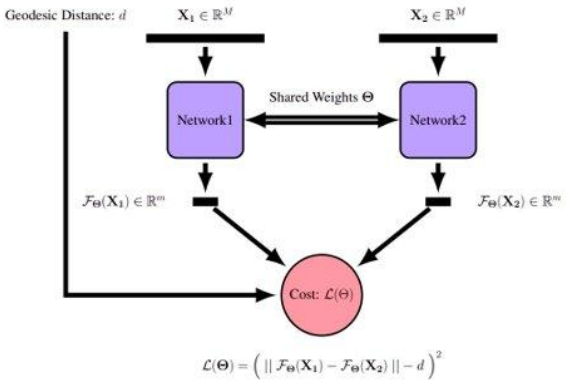


Figura 2: Arquitetura Deep Isometric Manifold Learning (R. Kimmel, 2019, #)

➢ Deep Manifold Transformation: Utilização de autoencoder de reconstrução, Local Geometry Preserving Constraints: Constituem a loss da rede

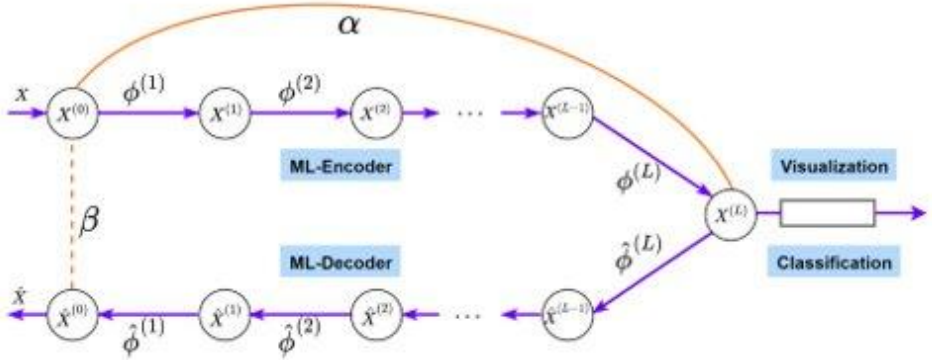


Figura 3: Arquitetura Deep Manifold Transformation

➢ Generalized Clustering and Multi-Manifold Learning.

No link pode ser encontrada uma lista com os artigos levantados para o estudo de deep manifold learning, este trabalho aborda especificamente os algoritmos relacionados com autoencoders.

2.3 Autoencoders

Um autoencoder é um tipo especial de rede neural treinada para copiar sua entrada para sua saída. Por exemplo, dada uma imagem de um dígito manuscrito, um autoencoder primeiro codifica a imagem em uma representação latente de dimensão inferior e, em seguida, decodifica a representação latente de volta para uma imagem. Um autoencoder aprende a compactar os dados enquanto minimiza o erro de reconstrução. Os autoencoders se enquadram em algoritmos de aprendizado não supervisionados, pois aprendem a representação compacta dos dados automaticamente a partir dos dados de entrada sem rótulos.

2.3.1 Principais aplicações dos Autoencoders

Hoje, as duas aplicações práticas de autoencoders são redução de ruído de dados e redução de dimensionalidade para visualização de dados. Com dimensionalidade apropriada e restrições de esparsidade, os autoencoders podem aprender projeções de dados que são mais interessantes do que o PCA, T-SNE, e Umap ou outras técnicas básicas. (Pramoditha, n.d.).

A seguir são listadas as principais aplicações dos autoencoders:

- Compactar imagens para economizar memória (compactação de imagem)
- Reduzir a dimensionalidade dos dados (redução da dimensionalidade)
- Transformar dados ruidosos em dados limpos usando codificadores automáticos de redução de ruído (denoising)
- Adicionar cor a imagens em tons de cinza (colonização de imagem) usando autoencoders de colonização automática
- Aumentar a resolução das imagens para melhorar os detalhes (super-resolução)
- Extrair os recursos mais importantes dos dados de entrada (extração de recursos)
- Gerar novas imagens com pequenas variações usando autoencoders variacionais (geração de imagens)
- Remover marcas d'água de imagens (remoção de marca d'água)

2.3.2 Elementos-chave de um autoencoder

Um autoencoder consiste nos seguintes elementos.

Codificador: Esta é geralmente uma função não linear que transforma a entrada, x, em um vetor latente de menor dimensão que é denotado por z=f(x). Este processo é conhecido como codificação.

Vetor latente: Isso também é referido como representação latente ou código latente em alguma literatura. Isso representa os recursos mais importantes dos dados de entrada em uma forma de dimensão inferior. Esta é a saída fornecida pelo codificador e será a entrada para o decodificador. A dimensão do vetor latente pode ser maior que x. Nesse caso, o autoencoder simplesmente copiará a entrada para a saída. Em outras palavras, ele tentará memorizar os dados de entrada. No entanto, na maioria dos casos, a dimensão do vetor latente é muito menor que x.

Decodificador: Esta também é uma função não linear que usa o vetor latente z, como entrada e emite uma representação compactada (entrada recuperada) dos dados de entrada. Este processo é conhecido como decodificação e é matematicamente denotado por $g(z)=\bar{x}$. O decodificador só pode aproximar a entrada porque o vetor latente é de baixa dimensão. Então, x não é exatamente o mesmo que $\bar{x}$.

A figura 4 apresenta os 3 componentes principais de um autoencoder, o codificador, código e decodificador. O codificador comprime a entrada e produz o código, o decodificador então reconstrói a entrada usando apenas este código.

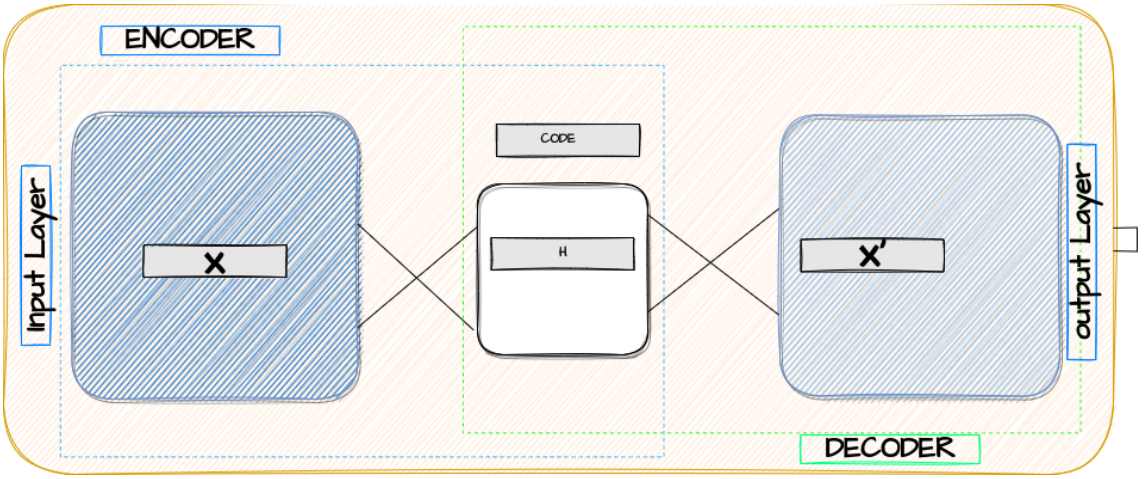


Figura 4*:* Componentes principais de um autoencoder, o codificador, código e decodificador.

Função de perda: Uma função de perda é usada para medir a dissimilaridade entre a entrada, x, e a saída, x'. Podemos usar erro quadrático médio (MSE) ou entropia cruzada binária como a função de perda. Ao treinar um autoencoder, o objetivo é tornar a entrada recuperada, x', o mais próximo possível da entrada original, x. Para isso, devemos tentar minimizar a função de perda durante o treinamento. A função de perda deve ser diferenciável como requisito para a retropropagação.

2.3.4 Tipos de Autoencoders

Uma taxonomia de EAs pode ser construída de acordo com diferentes critérios. Seguindo as propriedades do modelo inferido em relação ao recurso tarefa, o artigo (Charte & Charte, 2018) divide em quatro categorias principais nesta taxonomia:

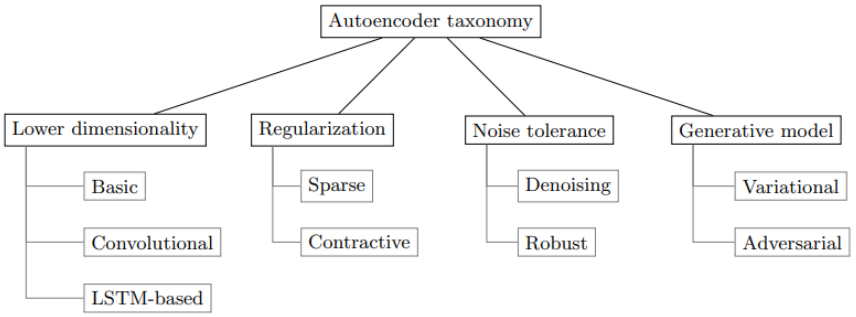


Figura 5:Taxonomia de autoencoders baseada na tarefa (Charte & Charte, 2018)

A continuação é apresentado o detalhamento de alguns tipos de autoencoders abordados neste trabalho para a resolução do objetivo de redução de dimensionalidade seguindo como referência o (*Capítulo 59 - Principais Tipos De Redes Neurais Artificiais Autoencoders*, n.d.)

Autoencoder Padrão

Na sua forma mais simples, o Autoencoder é uma rede neural artificial de três camadas, isto é, uma rede neural com uma camada de entrada, uma oculta e uma camada de saída. A entrada e a saída são as mesmas e aprendemos a reconstruir a entrada, por exemplo, usando o otimizador adam e a função de perda de erro quadrático médio.

Autoencoder Multicamada

Se uma camada oculta não for suficiente, obviamente podemos estender o Autoencoder para mais camadas ocultas. Nossa implementação poderia usar 3 camadas ocultas em vez de apenas uma. Qualquer uma das camadas ocultas pode ser escolhida como representação de recurso, mas o ideal é tornar a rede simétrica e usar a camada mais intermediária.

Autoencoder Convolucional

Os Autoencoders podem ser usados com convoluções em vez de camadas totalmente conectadas mas usando imagens (vetores 3D) em vez de vetores 1D achatados (flattened). A imagem de entrada é reduzida para fornecer uma representação latente de dimensões menores e forçar o Autoencoder a aprender uma versão compactada das imagens.

Autoencoder Regularizado

Existem outras maneiras pelas quais podemos restringir a reconstrução de um Autoencoder, além de impor uma camada oculta de menor dimensão que a entrada. Em vez de limitar a capacidade do modelo mantendo o codificador e o decodificador rasos e o tamanho do código pequeno, os Autoencoders regularizados usam uma função de perda que incentiva o modelo a ter outras propriedades além da capacidade de copiar sua entrada para sua saída. Na prática, geralmente encontramos dois tipos de Autoencoder Regularizado: o Autoencoder Esparso e o Autoencoder Denoising.

2.3.5 Vantagens dos Autoencoders sobre o UMAP e o T-NSE (Frenzel, 2020)

O Autoencoders tem várias vantagens exclusivas sobre os métodos de aprendizado múltiplos comuns, como t-SNE e UMAP:

Em vez de apenas uma transformação dos dados de treinamento, ele fornece uma função reversível e determinística, mapeando do espaço de dados para o espaço de incorporação.

Devido à reversibilidade do mapeamento, o modelo pode ser usado para gerar novos dados a partir de variáveis latentes arbitrárias. Também os torna altamente adequados como representações intermediárias para tarefas a jusante.

Depois que um modelo é treinado, ele pode ser reutilizado para transformar novos dados, tornando-o adequado para uso em configurações ao vivo.

Como o UMAP, o CVAE é rápido e escala muito melhor para grandes conjuntos de dados, entrada de alta dimensão e espaços latentes.

A arquitetura da rede neural e os parâmetros de treinamento são altamente personalizáveis por meio da API simples, permitindo que usuários mais avançados adaptem o sistema às suas necessidades.

Os VAEs têm uma base teórica muito forte e os espaços latentes aprendidos têm muitas propriedades desejáveis. Há também extensa literatura sobre diferentes variantes, e o CVAE pode ser facilmente estendido para acompanhar os novos avanços da pesquisa.

2.4 Identificação de atividades humanas (HAR)

A ideia principal dos sistemas de identificação de atividades humanas é utilizar uma combinação de dados coletados via dispositivos vestíveis e sensores de smartphones ou de imagens e vídeos para identificar corretamente atividades primárias como, "correr", "dormir", etc, bem como outras informações complementares (Jobanputra, Bavishi, and Doshi 2019).

2.4.1 MotionSense (Malekzadeh et al., n.d.)

Este conjunto de dados inclui dados de séries temporais gerados por sensores de acelerômetro e giroscópio. É coletado com um iPhone 6s guardado no bolso frontal do participante usando o SensingKit que coleta informações do framework Core Motion em dispositivos iOS. Um total de 24 participantes em uma variedade de sexo, idade, peso e altura realizaram 6 atividades em 15 tentativas no mesmo ambiente e condições: andar de baixo, andar de cima, andar, correr, sentar e ficar em pé.

2.4.2 Kuhar

Este conjunto de dados contém informações sobre 18 atividades diferentes coletadas de 90 participantes (75 homens e 15 mulheres) usando sensores de smartphones (acelerômetro e giroscópio). Possui 1.945 amostras brutas de atividades coletadas diretamente dos participantes e 20.750 sub amostras extraídas deles. (*KU-HAR: An Open Dataset for Human Activity Recognition*, 2021)

2.4.3 Dataset Extrasensory

Extrasensory (Vaizman et al. 2018) é um dataset público formado por dados de sensores de smartphones e smartwatches, cuja particularidade é a captura de dados em um ambiente não controlado.
As atividades humanas, descritas pelos rótulos de cada amostra do dataset, foram divididas em dois grupos:
Atividade principal: classes mutuamente exclusivas Lying down, sitting, standing in place, standing and moving, walking, running, bicycling.
Atividade secundária. 109 rótulos adicionais que descrevem um contexto mais específico em diferentes aspectos: esportes (por exemplo, jogar basquete, na academia), transporte (por exemplo, dirigir - eu sou o motorista, no ônibus), necessidades básicas (por exemplo, dormir, comer, ir ao banheiro), companhia (por exemplo, com a família, com colegas de trabalho), localização (por exemplo, em casa, no trabalho, fora) etc.

3. Metodologia e resultados

Inicialmente se fez uma exploração dos algoritmos autoencoders usando bases de dados conhecidas e finalmente foi estudado e viabilizada a implantação com os datasets.

Este capítulo descreve a implementação dos algoritmos Autoencoders. Nos item 3.1 e 3.2 se detalham os experimentos realizados com o dataset comum "digits e mnist" (*MNIST Handwritten Digit Database*, n.d.). O objetivo com essas implementações é entender o funcionamento dos Autoencoders usando tensorflow (Dibia, n.d.) e pythorch (*Implementing an Autoencoder in PyTorch*, 2022).

No item 3.3 se detalha os experimentos realizados sobre os datasets HAR: motionsense, kuhar e dados dados próprios capturados no device. O objetivo é fazer comparações de performance e precisão entre os diferentes modelos gerados com os algoritmos autoencoders e o algoritmo umap.

3.1 Implementação dos autoencoders usando tensorflow

Nesta seção é apresentada a implentação de quatro tipos de autoencoders: o *TFBasicAutoencoder, TFSparsityConstraintAutoencoder, DeepAutoencoder E Convolutional Autoencoder.* O codigo fonte pode ser encontrado aqui.

Os experimentos foram feitos usando o banco de dados MNIST de dígitos manuscritos, o qual possui um conjunto de treinamento de 60.000 exemplos e um conjunto de testes de 10.000 exemplos.

3.1.1 Autoencoder básico

A imagem 1 apresenta a implementação básica de um autoencoder com duas camadas densas: um codificador, que comprime as imagens em um vetor latente de 64 dimensões, e um decodificador, que reconstrói a imagem original a partir do espaço latente usando a API Keras.

```
"""Vamos espremer o espaço de N dimensões em um espaço latente de n dimensões.
 A arquitetura do modelo Autoencoder são muito rudimentares, o objetivo é demonstrar como um codificador funciona."""

from sklearn.metrics import accuracy_score, precision_score, recall_score
from sklearn.model_selection import train_test_split
from tensorflow.keras import layers, losses
from tensorflow.keras.models import Model
import tensorflow as tf
latent_dim = 64

class Autoencoder(Model):
  def __init__(self, latent_dim):
    super(Autoencoder, self).__init__()
    self.latent_dim = latent_dim
    self.encoder = tf.keras.Sequential([
      layers.Flatten(),
      layers.Dense(latent_dim, activation='relu'),
    ])
    self.decoder = tf.keras.Sequential([
      layers.Dense(latent_dim, activation='sigmoid'),
      layers.Reshape((28, 28))
    ])

  def call(self, x):
    encoded = self.encoder(x)
    decoded = self.decoder(encoded)
    return decoded
```

Imagem 1: Implementação básica de um autoencoder com duas camadas densas

O autoencoder básico vai ser treinado usando o conjunto de dados Fashion MNIST. Cada imagem neste conjunto de dados tem 28 x 28 pixels.

```
[7]: #2) carregando o dataset fashion_mnist
     (x_train, _), (x_test, y_test) = fashion_mnist.load_data()

     x_train = x_train.astype('float32') / 255.
     x_test = x_test.astype('float32') / 255.

     print (x_train.shape)
     print (x_test.shape)

     (60000, 28, 28)
     (10000, 28, 28)
```

O modelo foi treinado usando x_train como entrada e destino. O codificador aprenderá a comprimir o conjunto de dados de 784 dimensões para o espaço latente e o decodificador aprenderá a reconstruir as imagens originais.

```
#Train the model using x_train as both the input and the target.
autoencoder = Autoencoder(latent_dim)
autoencoder.compile(optimizer='adam', loss=losses.MeanSquaredError())
autoencoder.fit(x_train, x_train,
                epochs=10,
                shuffle=True,
                validation_data=(x_test, x_test))
```

Depois do que o modelo esteja treinado pode ser testada a codificação e decodificando imagens do conjunto de teste.

```
encoded_imgs = autoencoder.encoder(x_test).numpy()
decoded_imgs = autoencoder.decoder(encoded_imgs).numpy()
print("encoded_imgs->",encoded_imgs.shape)
print("decoded_imgs->",decoded_imgs.shape)
```

```
encoded_imgs-> (10000, 64)
decoded_imgs-> (10000, 28, 28)
```

3.1.2 Adicionando uma restrição de esparsidade nas representações codificadas

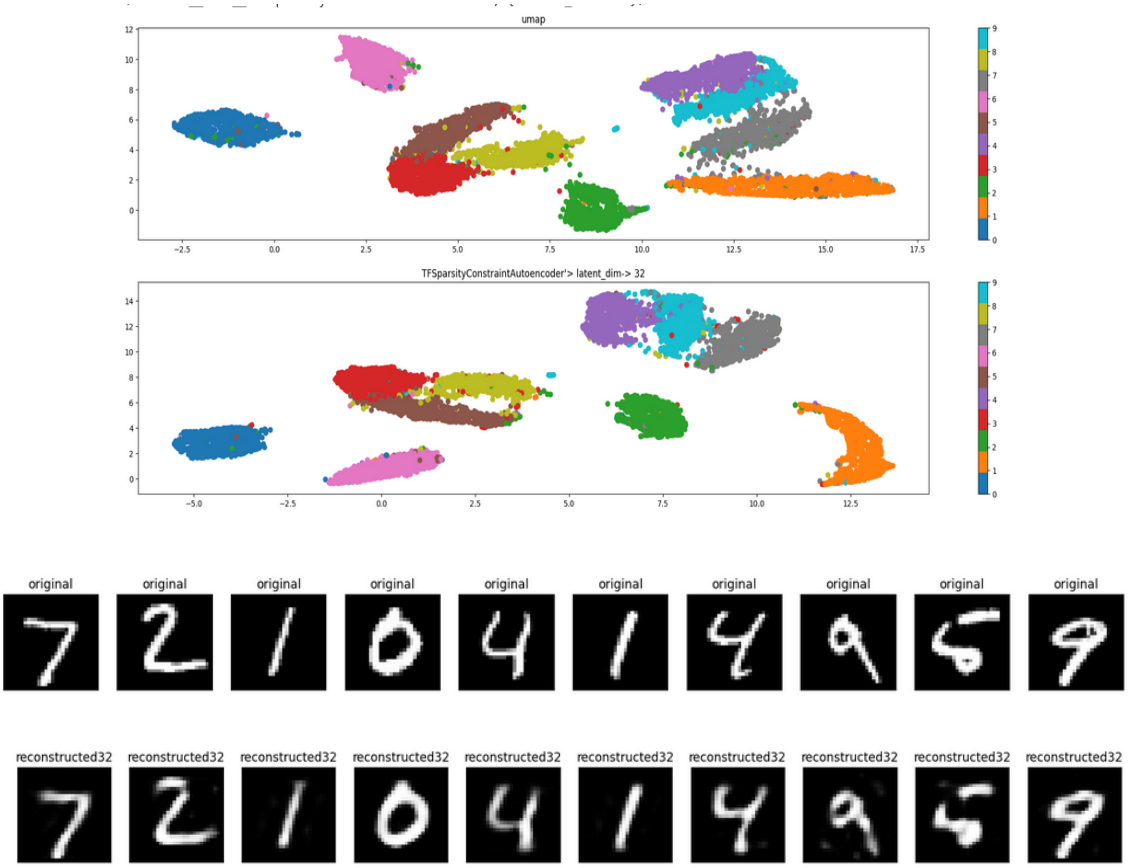
No exemplo 3.1.1, as representações foram limitadas apenas pelo tamanho da camada oculta (64). Em tal situação, o que normalmente acontece é que a camada oculta está aprendendo uma aproximação de PCA (análise de componentes principais). Mas outra maneira de restringir as representações para serem compactas é adicionar uma restrição de esparsidade na atividade das representações ocultas, para que menos unidades "disparem" em um determinado momento.
No Keras, isso pode ser feito adicionando um *activity_regularizer* na camada Dense:

```
class TFSparsityConstraintAutoencoder(Model):
  def __init__(self, latent_dim):
    super(TFSparsityConstraintAutoencoder, self).__init__()
    self.latent_dim = latent_dim
    self.encoder = tf.keras.Sequential([
      layers.Dense(latent_dim, activation='relu',activity_regularizer=regularizers.l1(10e-5)),
    ])
    self.decoder = tf.keras.Sequential([
      layers.Dense(784, activation='sigmoid'),
    ])

  def call(self, x):
    encoded = self.encoder(x)
    decoded = self.decoder(encoded)
    return decoded
```

O modelo foi treinado por 100 épocas (com a regularização adicionada, o modelo tem menos probabilidade de super ajustar e pode ser treinado por mais tempo). Os modelos terminam com uma perda de trem de 0,125 e perda de teste de 0,124. A diferença entre os dois deve-se principalmente ao termo de regularização ser adicionado à perda durante o treinamento (no valor de cerca de 0,01).

```
Epoch 100/100
1875/1875 [==============================] - 3s 2ms/step - loss: 0.0125 - val_loss: 0.0124
```

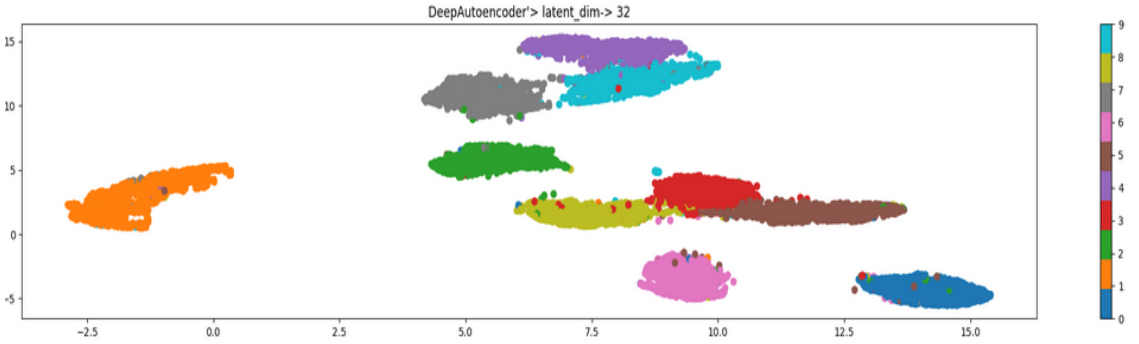
Aqui está uma visualização de nossos novos resultados:



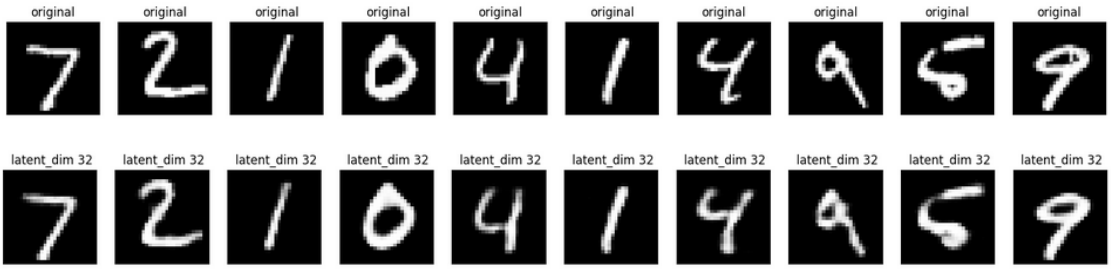
Eles se parecem bastante com o modelo anterior, sendo a única diferença significativa a dispersão das representações codificadas. encoded_imgs.mean() produz um valor 3,33 (sobre nossas 10.000 imagens de teste), enquanto com o modelo anterior a mesma quantidade era 7,30. Portanto, nosso novo modelo produz representações codificadas que são duas vezes mais esparsas.

3.1.3 Deep autoencoder

Não precisamos nos limitar a uma única camada como codificador ou decodificador, podemos usar uma pilha de camadas, como:



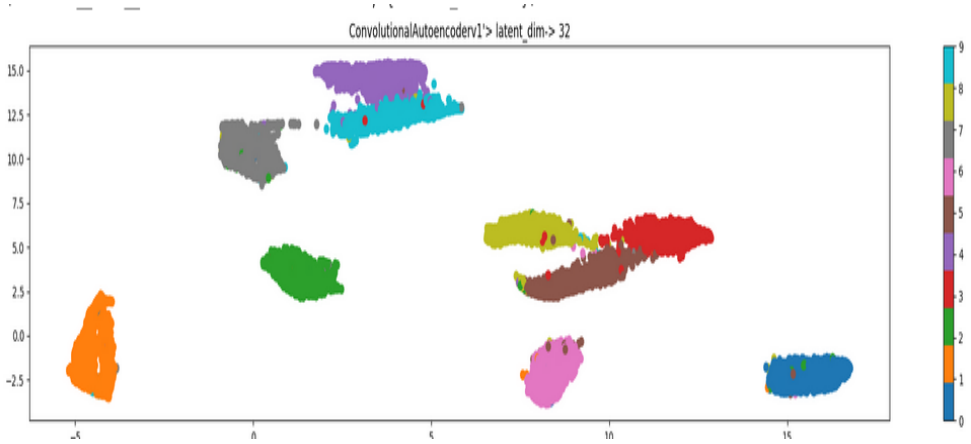
Após 50 épocas, atinge uma perda de treinamento e validação de ~0,08, um pouco melhor do que os modelos anteriores. Os dígitos reconstruídos parecem um pouco melhores também:



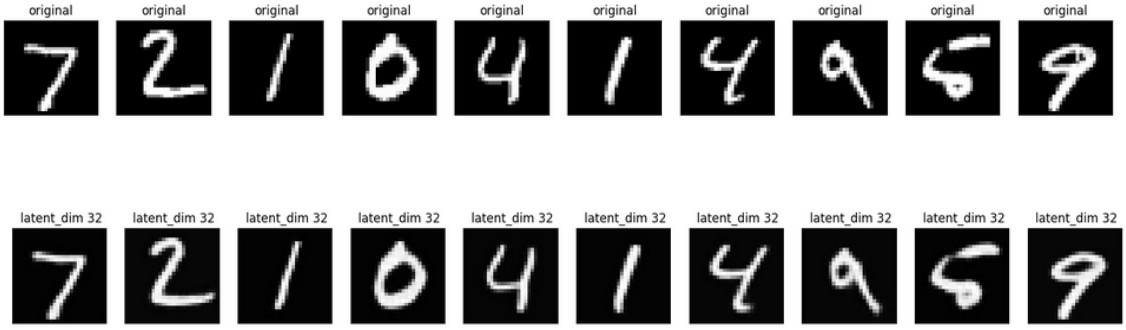
3.1.4 Autoencoder convolucional

O codificador consiste em uma pilha de camadas Conv2D e MaxPooling2D (o pool máximo sendo usado para redução da amostragem espacial), enquanto o decodificador consistirá em uma pilha de camadas Conv2D e UpSampling2D.

O modelo converge para uma perda de 0,094, significativamente melhor do que os modelos anteriores (isso se deve em grande parte à maior capacidade entrópica da representação codificada, 128 dimensões contra 32 anteriormente).



Vamos dar uma olhada nos dígitos reconstruídos:

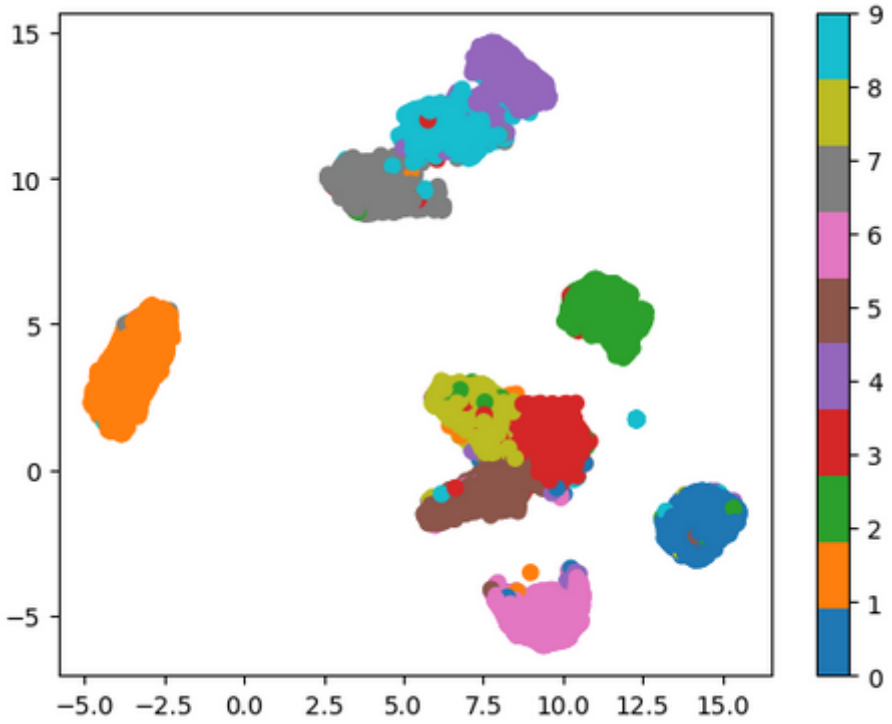


3.2 Implementação dos autoencoders usando pytorch

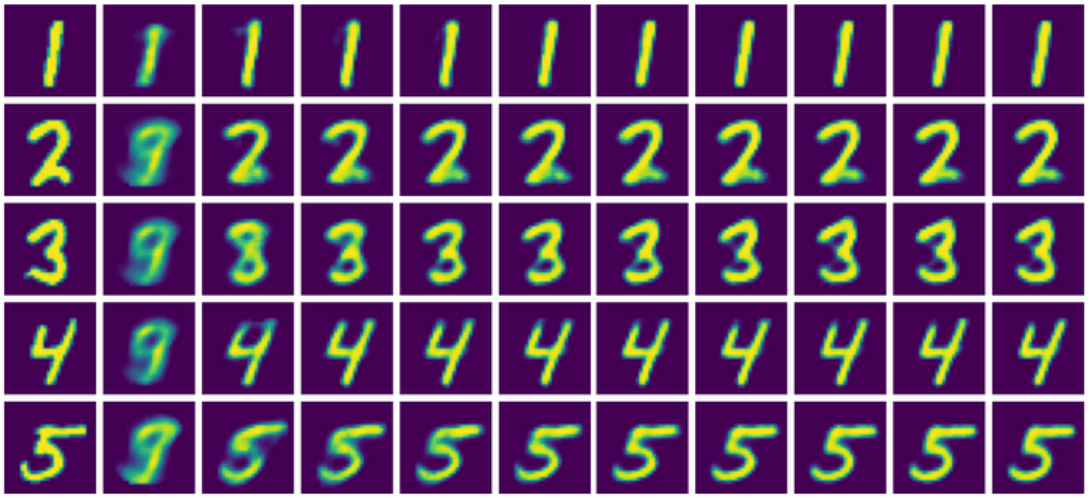
```
class Encoder(nn.Module):
    def __init__(self, latent_dims,input_size):
        input_=np.prod(input_size)
        print(input_)
        super(Encoder, self).__init__()
        self.linear1 = nn.Linear(28*28, 512)
        self.linear2 = nn.Linear(512, 256)
        self.linear3 = nn.Linear(256, 128)
        self.linear4 = nn.Linear(128, latent_dims)

    def forward(self, x):
        x = torch.flatten(x, start_dim=1) # it flattens the image x by transforming it to 1-d tensor
        x = F.relu(self.linear1(x))
        x = F.relu(self.linear2(x))
        x = F.relu(self.linear3(x))

        return self.linear4(x)
```

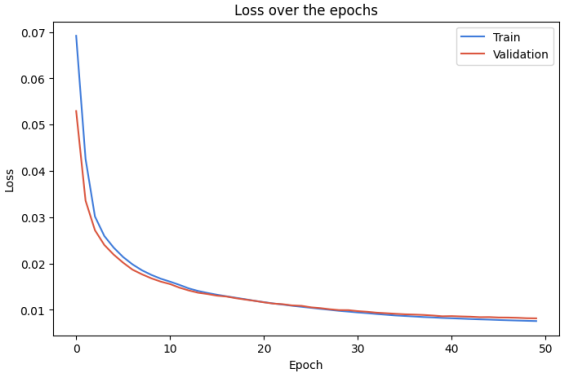
```
class Decoder(nn.Module):
    def __init__(self, latent_dims,input_size):
        super(Decoder, self).__init__()
        self.linear1 = nn.Linear(latent_dims, 128)
        self.linear2 = nn.Linear(128, 256)
        self.linear3 = nn.Linear(256, 512)
        self.linear4 = nn.Linear(512, 28*28)

    def forward(self, z):
        z = F.relu(self.linear1(z))
        z = F.relu(self.linear2(z))
        z = F.relu(self.linear3(z))
        z = torch.sigmoid(self.linear4(z))

        return z.reshape((-1, 1,28,28)) # it transforms the generated tensor into image


            class Autoencoder(nn.Module):
                def __init__(self, latent_dims,input_size):

                    super(Autoencoder, self).__init__()
                    self.encoder = Encoder(latent_dims,input_size)
                    self.decoder = Decoder(latent_dims,input_size)

                def forward(self, x):
                    z = self.encoder(x)
                    return self.decoder(z)


def train(model, train_loader, val_loader, epochs=20):
    log_dict = {"train_loss": [],
                "val_loss": [],
                "reconstructed_images":[]}
    opt = torch.optim.Adam(model.parameters(), lr=3e-4)
    for epoch in tqdm.tqdm_notebook(range(epochs)):
        train_loss = []
        model.train() # set model to training mode
        for x, y in train_loader:
            x = x.to(device) # GPU

            opt.zero_grad()
            x_hat = model(x)

            loss = ((x - x_hat)**2).mean() # reconstruction loss
            train_loss.append(loss.to('cpu').detach().numpy())

            loss.backward() # compute gradients
            opt.step() # update weights

        val_loss = []
        model.eval() # set model to evaluation mode
        with torch.no_grad():
            for x, y in val_loader:
                x = x.to(device) # GPU
                x_hat = model(x)

                loss = ((x - x_hat)**2).mean() # reconstruction loss
                val_loss.append(loss.to('cpu').detach().numpy())

        log_dict["train_loss"].append(np.array(train_loss).mean())
        log_dict["val_loss"].append(np.array(val_loss).mean())
        log_dict["reconstructed_images"].append(x_hat.to('cpu').detach()[:5])
    log_dict["original_images"] = x.to('cpu').detach()[:5]

    return model, log_dict
```

Dos gráficos seguintes podemos concluir que a implementação dos resultados para o dataset mnist são bons, podemos ver a separação das classes ainda com conflitos entre o 3,5,e 8.



O seguinte gráfico apresenta os resultados das imagens geradas ao longo das épocas para os números 1,2,3,4 e 5 (coluna 1), a partir da quarta época os pixels das imagens são similares ao original.





3.3 implementações usando os datasets HAR

Vamos comparar não apenas a performance na representação dos dados mas também no tempo de execução para que o algoritmo processe os dados usando diferentes conjuntos descritos no item 2.4 deste documento.

As implementações de redução de dimensionalidade utilizando autoencoders se encontram no notebook disponibilizado no link.

Os autoencoders descritos no item 3.1 deste documento foram customizados e adequados para os datasets HAR como é apresentado na figura 6.

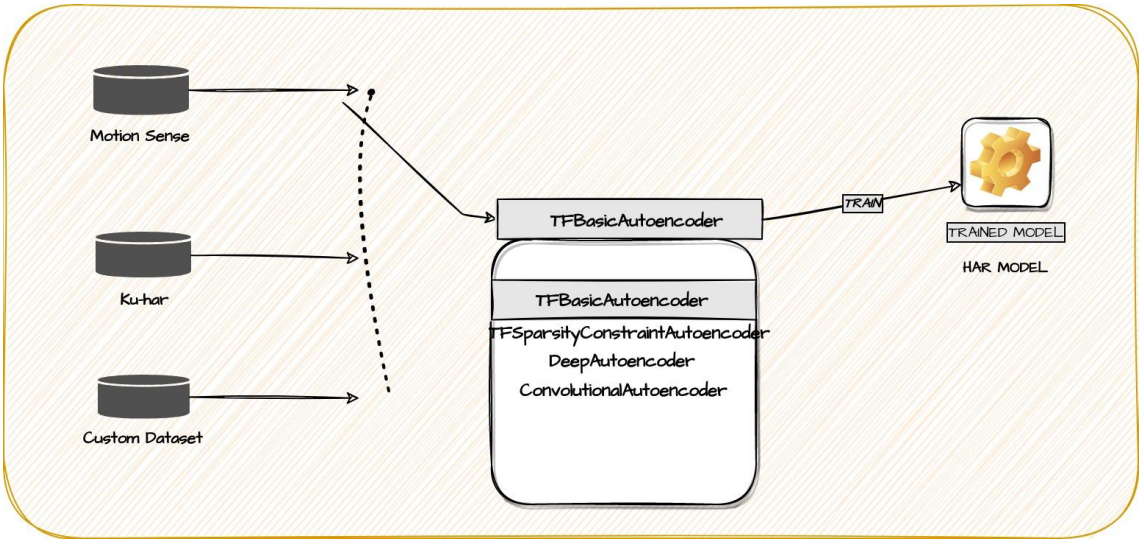


Figura 6: Experimentos com datasets HAR e Autoencoders

Foi implementada uma função **train_datasets** que permite fazer o treinamento e comparação dos diferentes tipos de autoencoders.

```python
def train_datasets(reducers,datasets):

    for dataset in datasets:
        data_encoded = []
        data_decoded = []
        loss= []
        titles= []


        for reducer, args in reducers:
            x_train, y_train,x_test,y_test,input_shape=load_dataset(dataset)
            if(reducer==ConvolutionalAutoencoder):
                x_train,x_test=dataset_reshape(x_train,x_test,dataset)
                input_shape=(x_train.shape[1],x_train.shape[2],x_train.shape[3])
            print(dataset, x_train.shape, y_train.shape,x_test.shape,y_test.shape,input_shape)
            #Train the model using x_train as both the input and the target.
            autoencoder = reducer(latent_dim=args["latent_dim"],input_shape=input_shape)
            autoencoder.compile(optimizer='adam', loss = tf.keras.losses.MeanSquaredLogarithmicError())
            #autoencoder.encoder.summary()
            #autoencoder.decoder.summary()
            hist=autoencoder.fit(x_train, x_train,
                        epochs=args['epochs'],
                        shuffle=True,
                        validation_data=(x_test, x_test))
            start_time = time.time()
            encoded_imgs = autoencoder.encoder(x_test).numpy()
            elapsed_time = time.time() - start_time
            decoded_imgs = autoencoder.decoder(encoded_imgs).numpy()
           # print("encoded_imgs->",encoded_imgs.shape)
           # print("decoded_imgs->",decoded_imgs.shape)
            data_encoded.append(encoded_imgs)
            data_decoded.append(decoded_imgs)
            loss.append(hist.history['loss'])
            titles.append(str(reducer).split('__.')[1] +" latent_dim= "+
                          str(args["latent_dim"]) + ' time='+"{:.4f} s".format(elapsed_time))

        plot_scatter(x_test,y_test,dataset,data_encoded,titles)
        plot_loss(loss,titles,dataset)
```

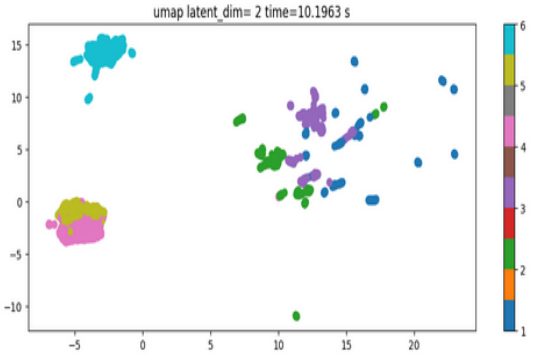
3.3.1 Motionsense

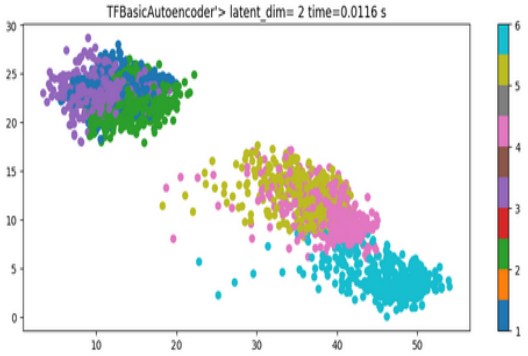
Foram feitos testes para o dataset Motionsense usando quatro tipos de autoencoders e diferentes valores de dimensões de redução (2,5,10,16,32).

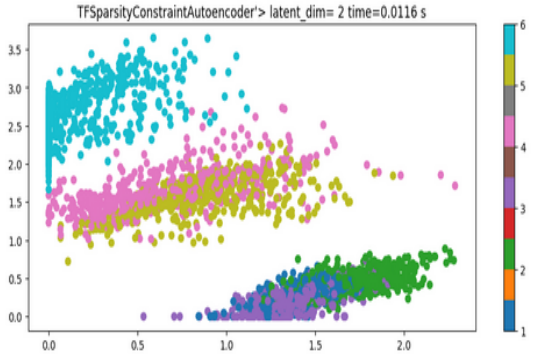
```
### dataset Motionsense

reducers = [
    (TFBasicAutoencoder, { "latent_dim": 2,'epochs': 50}),
   (TFSparsityConstraintAutoencoder, { "latent_dim": 2,'epochs': 100}),
    (DeepAutoencoder, { "latent_dim": 2,'epochs': 100}),
    (ConvolutionalAutoencoder, { "latent_dim": 2,'epochs': 200}),
    (TFBasicAutoencoder, { "latent_dim": 5,'epochs': 50}),
 (TFSparsityConstraintAutoencoder, { "latent_dim": 5,'epochs': 100}),
     (DeepAutoencoder, { "latent_dim": 5,'epochs': 100}),
     (ConvolutionalAutoencoder, { "latent_dim": 5,'epochs': 200}),
     (TFBasicAutoencoder, { "latent_dim": 10,'epochs': 50}),
   (TFSparsityConstraintAutoencoder, { "latent_dim": 10,'epochs': 100}),
      (DeepAutoencoder, { "latent_dim": 10,'epochs': 100}),
      (ConvolutionalAutoencoder, { "latent_dim": 10,'epochs': 200}),
       (TFBasicAutoencoder, { "latent_dim": 16,'epochs': 50}),
     (TFSparsityConstraintAutoencoder, { "latent_dim": 16,'epochs': 100}),
      (DeepAutoencoder, { "latent_dim": 16,'epochs': 100}),
    (ConvolutionalAutoencoder, { "latent_dim": 16,'epochs': 200}),
     (TFBasicAutoencoder, { "latent_dim": 32,'epochs': 50}),
   (TFSparsityConstraintAutoencoder, { "latent_dim": 32,'epochs': 100}),
    (DeepAutoencoder, { "latent_dim": 32,'epochs': 100}),
    (ConvolutionalAutoencoder, { "latent_dim": 32,'epochs': 200}),
]

datasets= ['Motionsense']

,#'custom','har'kuhar
train_datasets(reducers,datasets)
```

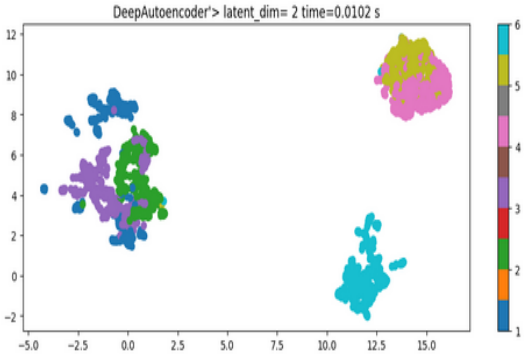
Dos resultados obtidos para o dataset motionsense, com redução de 2, pode se notar que o autoencoder convolutional foi o que apresentou melhor resultado, Fazendo a comparação com o umap todos os tipos de autoencoder apresentaram melhor tempo de execução no momento de ser fazer gerar os embeddings e visualmente a agrupação entre classes melhorou consideravelmente com os autoencoders.
As atividades deste dataset são listadas a continuação
1->WALKING, 2 ->WALKING_UPSTAIRS,3->3 WALKING_DOWNSTAIRS, 4->SITTING, 5->STANDING, 6->LAYING

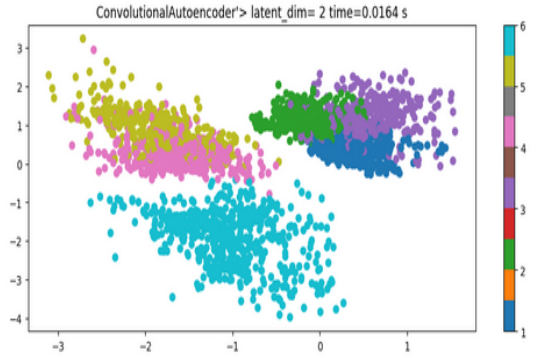
Dataset->Motionsense



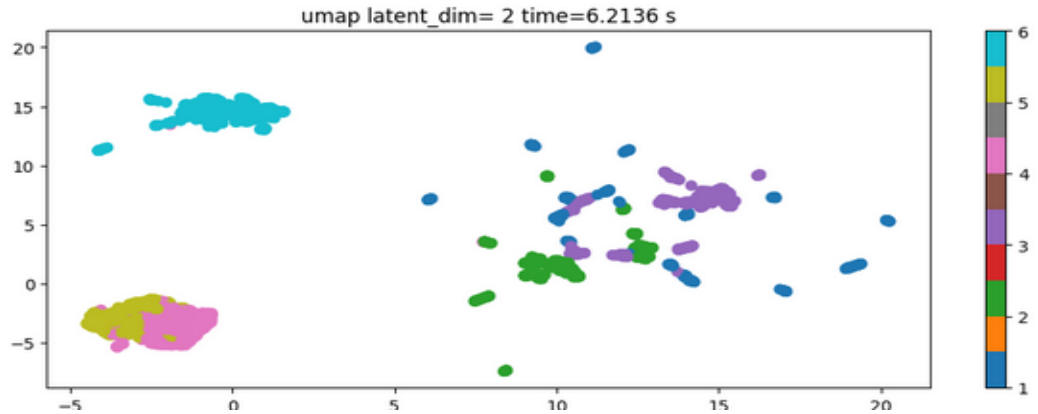


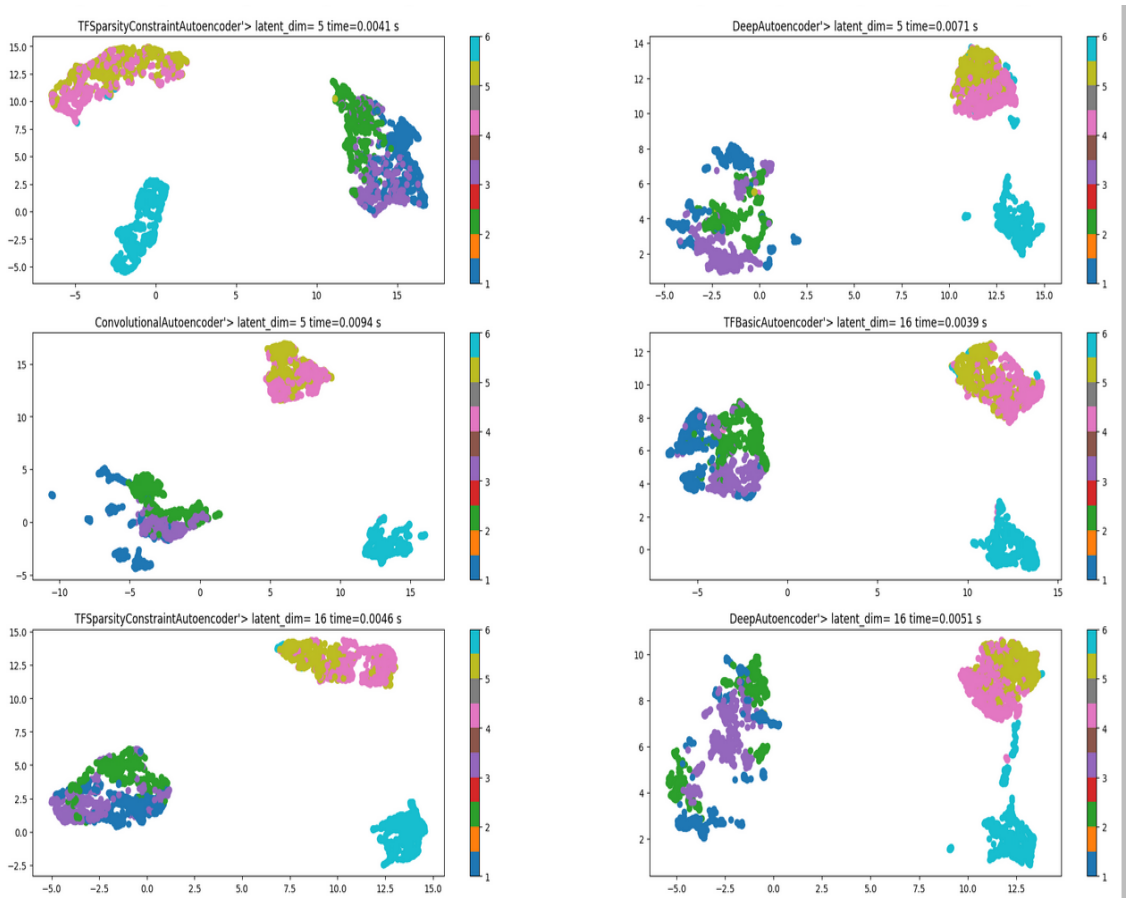






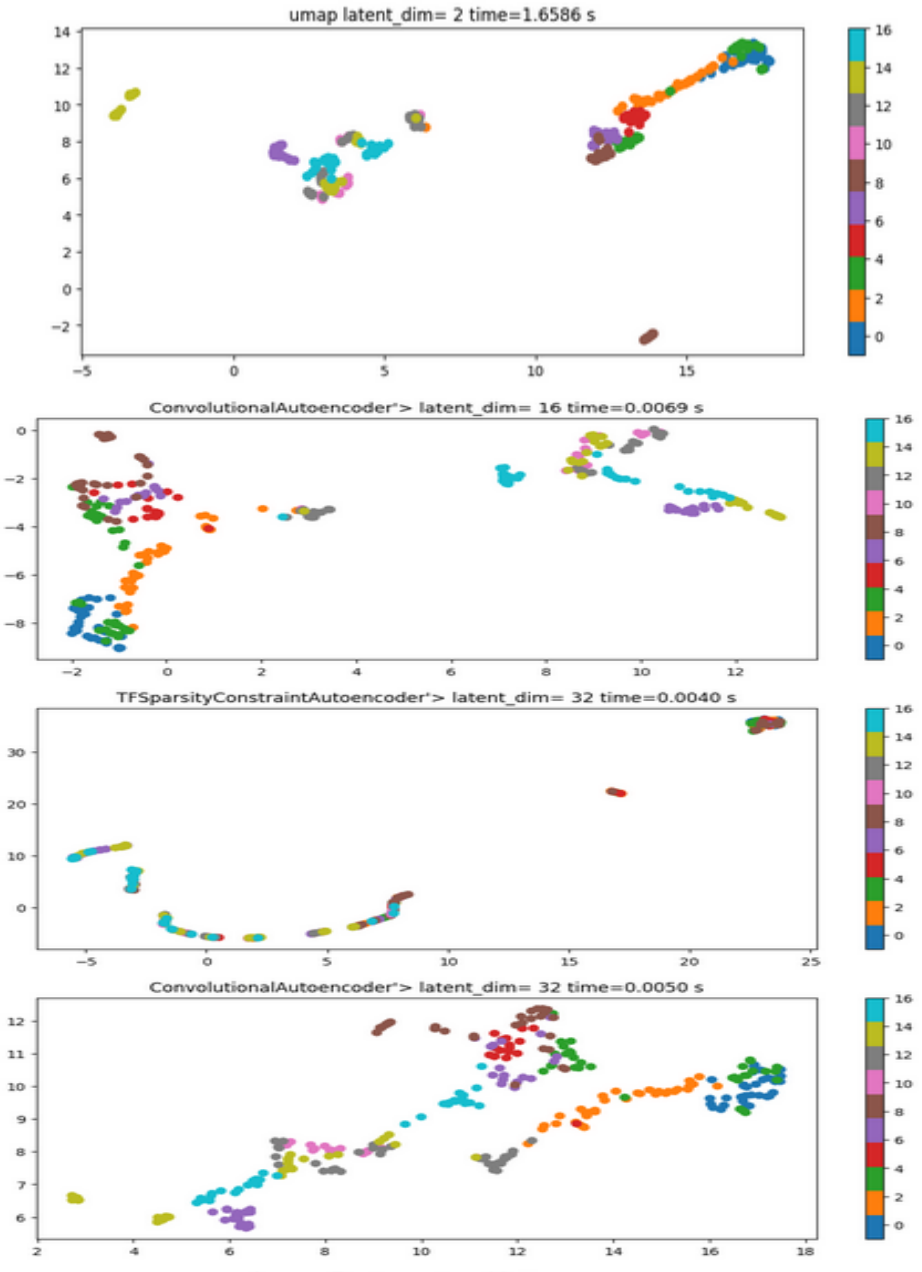
Dos seguintes gráficos podemos ver que os resultados melhoram ao utilizar os autoencoders mas ainda não se consegue a separação total entre atividades. Os testes vão ser continuados na seguinte fase para obter melhores resultados.

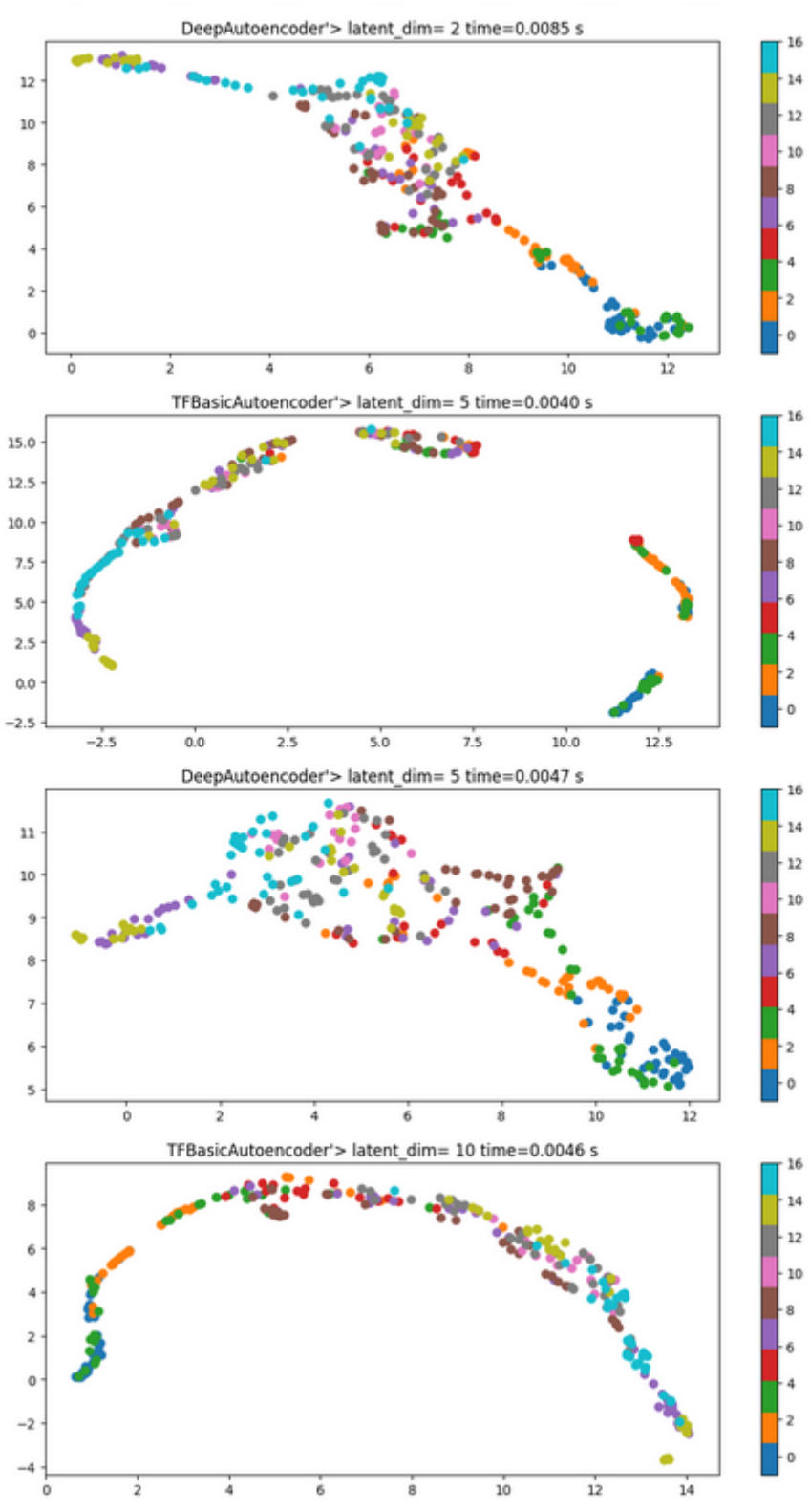


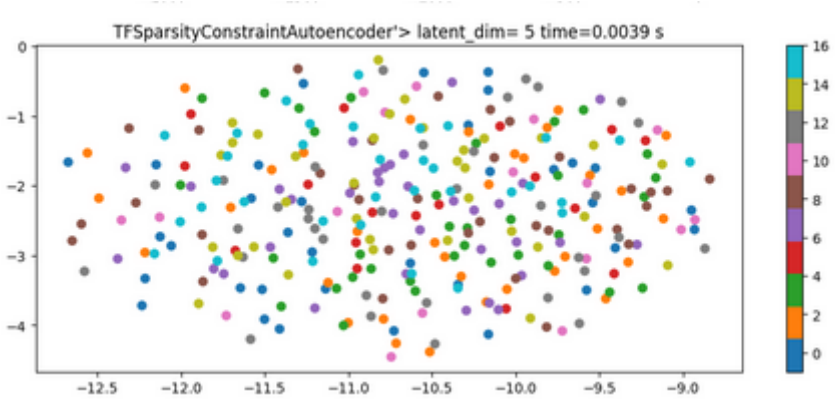


3.3.2 Kuhar

Os mesmos experimentos foram realizados para o dataset kuhar, dos gráficos podemos concluir que os melhores resultados foram com os autoencoders convolucionais, e os piores resultados foram para o *TFsparcityConstrain* de dimensão 5.

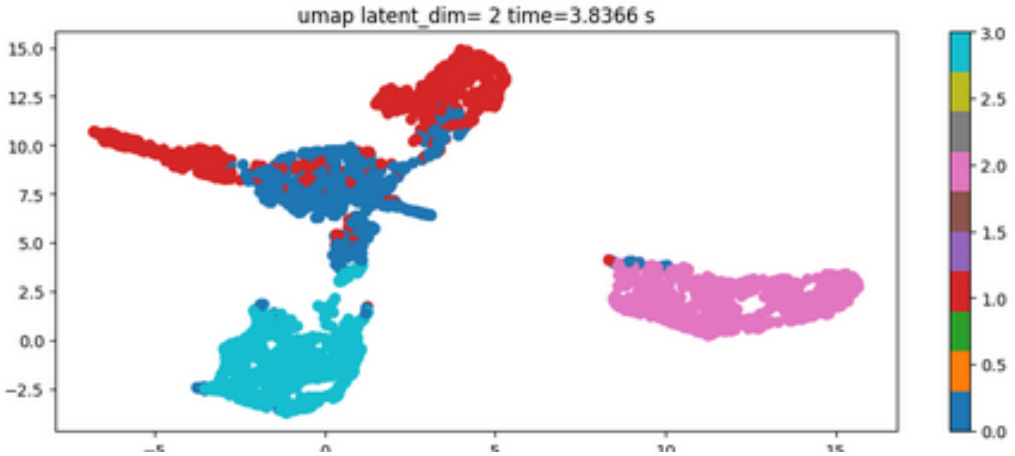


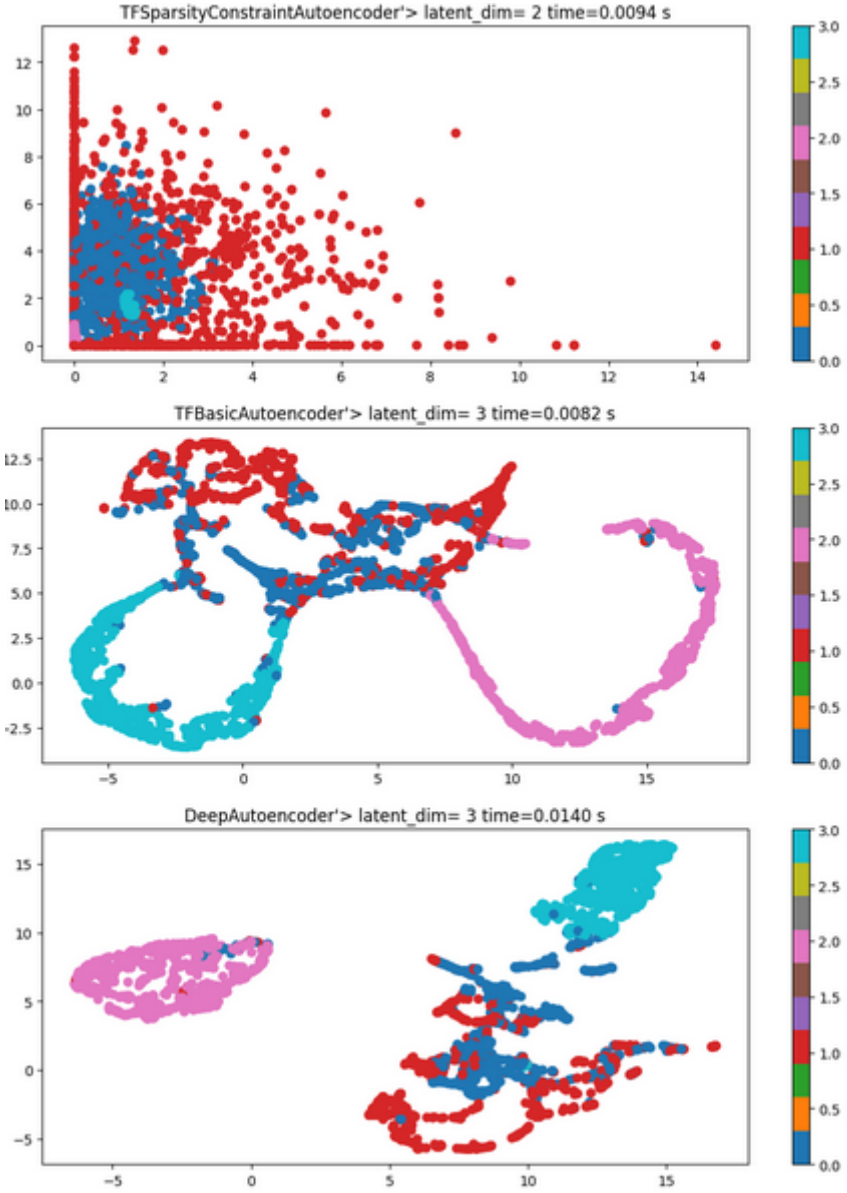

DeepAutoencoder'> latent_dim= 2 time=0.0085 s
TFBasicAutoencoder'> latent_dim= 5 time=0.0040 s
DeepAutoencoder'> latent_dim= 5 time=0.0047 s
TFBasicAutoencoder'> latent_dim= 10 time=0.0046 s

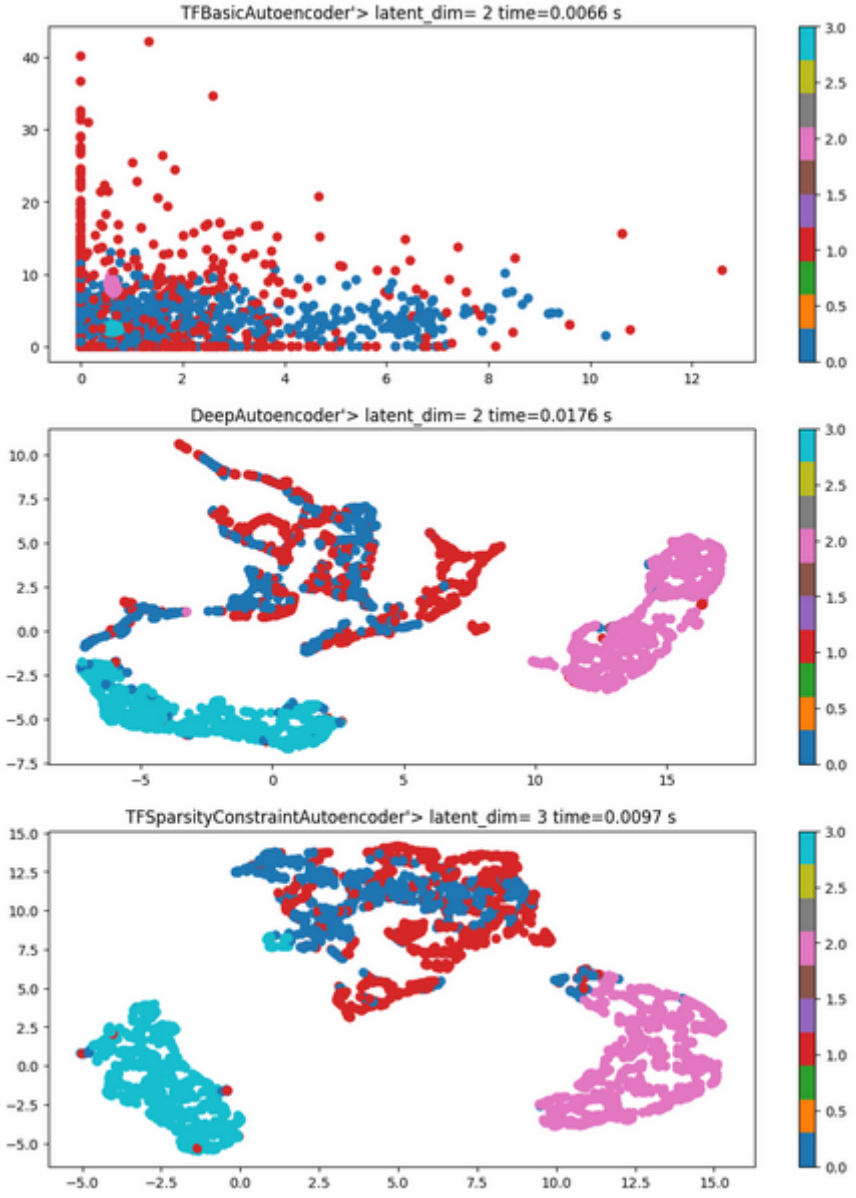


3.3.3 Dataset customizado

No dataset customizado os autoencoders não foram melhor do que o umap, vamos continuar os testes para verificar o motivo do resultado. o tempo demorado para fazer a redução foi menor utilizando o algoritmos autoencoderes.




TFSparsityConstraintAutoencoder'> latent_dim= 2 time=0.0094 s
TFBasicAutoencoder'> latent_dim= 3 time=0.0082 s
DeepAutoencoder'> latent_dim= 3 time=0.0140 s



3.4 Ferramentas para a implementação dos autoencoders

Existem algumas ferramentas prontas que permitem a implementação dos autoencoders, a seguinte lista apresenta algumas bibliotecas que serão exploradas na próxima atividade da meta.

- *Pythae*: é uma versátil biblioteca Python de código aberto que fornece uma implementação unificada e uma estrutura dedicada que permite o uso direto, reproduzível e confiável de modelos de autoencoder generativos.

  link da biblioteca: https://pypi.org/project/pythae/
  link do artigo: https://arxiv.org/abs/2206.08309

- *autoencoder: O repositório* contém as ferramentas necessárias para construir de forma flexível um autoencoder em pytorch. O principal objetivo deste kit de ferramentas é permitir a experimentação rápida e flexível com autoencoders convolucionais de uma variedade de arquiteturas.

  link da biblioteca: https://pypi.org/project/autoencoder/
  link do github:
  https://github.com/IljaManakov/ConvolutionalAutoencoderToolkit

- *Concrete Autoencoders:* é um tipo de autoencoder, permite a instalação de autoencoders usando keras.

  link da biblioteca: https://pypi.org/project/concrete-autoencoder/
  link do colab com exemplos:
  https://colab.research.google.com/drive/11NMLrmToq4bo6WQ_4WX5G4uIjBHyrzXd

4. Conclusões e próximos passos

- Nesta etapa foram implementadas diferentes estratégias usando algoritmos de redução de dimensionalidade especificamente Autoencoders.
- Dos resultados se conclui que os algoritmos Autoencoders parecem ser prometedores para a redução de dimensionalidade, apresentando como vantagem sobre o umap e o TSNE uma arquitetura da rede neural com parâmetros de treinamento altamente personalizáveis, permitindo adaptações ao sistema e as necessidades.
- Como o UMAP, os autoencoders são rápidos e de escala muito melhor para grandes conjuntos de dados, entrada de alta dimensão e espaços latentes.
- Os próximos passos são continuar com os testes no tensorflow, pytorch e outras ferramentas de implementação, avaliar outros datasets HAR como o extrasensory e verificar a viabilidade de implementação no Android.

5. Referências

Capítulo 59 - Principais Tipos de Redes Neurais Artificiais Autoencoders. (n.d.). Deep Learning Book. Retrieved November 30, 2022, from https://www.deeplearningbook.com.br/principais-tipos-de-redes-neurais-artificiais-autoencoders/

Charte, D., & Charte, F. (2018). A practical tutorial on autoencoders for nonlinear feature fusion: Taxonomy, models, software and guidelines. Information Fusion, 44, 78-96. 10.1016/j.inffus.2017.12.007

Dibia, V. (n.d.). Intro to Autoencoders. TensorFlow. Retrieved November 30, 2022, from https://www.tensorflow.org/tutorials/generative/autoencoder

Frenzel, M. (2020, March 2). CompressionVAE — A Powerful and Versatile Alternative to t-SNE and UMAP. Towards Data Science. Retrieved November 28, 2022, from https://towardsdatascience.com/compressionvae-a-powerful-and-versatile-alternative-to-t-sne-and-umap-5c50898b8696

Implementing an Autoencoder in PyTorch. (2022, July 7). GeeksforGeeks. Retrieved November 30, 2022, from https://www.geeksforgeeks.org/implementing-an-autoencoder-in-pytorch/

KU-HAR: An Open Dataset for Human Activity Recognition. (2021, February 16). KU-HAR: An Open Dataset for Human Activity Recognition - Mendeley Data. Retrieved November 30, 2022, from https://data.mendeley.com/datasets/45f952y38r/5

Malekzadeh, M., Clegg, R. G., Cavallaro, A., & Haddadi, H. (n.d.). MotionSense Dataset : Smartphone Sensor Data - HAR. Kaggle. Retrieved November 30, 2022, from https://www.kaggle.com/datasets/malekzadeh/motionsense-dataset

MNIST handwritten digit database. (n.d.). Yann LeCun. Retrieved November 30, 2022, from http://yann.lecun.com/exdb/mnist/

Pramoditha, R. (n.d.). An Introduction to Autoencoders in Deep Learning | by Rukshan Pramoditha | Medium. Rukshan Pramoditha. Retrieved November 28, 2022,from https://rukshanpramoditha.medium.com/an-introduction-to-autoencoders-in-deep-learning-ab5a5861f81e

6. Anexos

Anexo A:

Código computacional

https://github.com/H-IAAC/meta-4/releases/tag/1.0