Relatório 20 - Visão Computacional

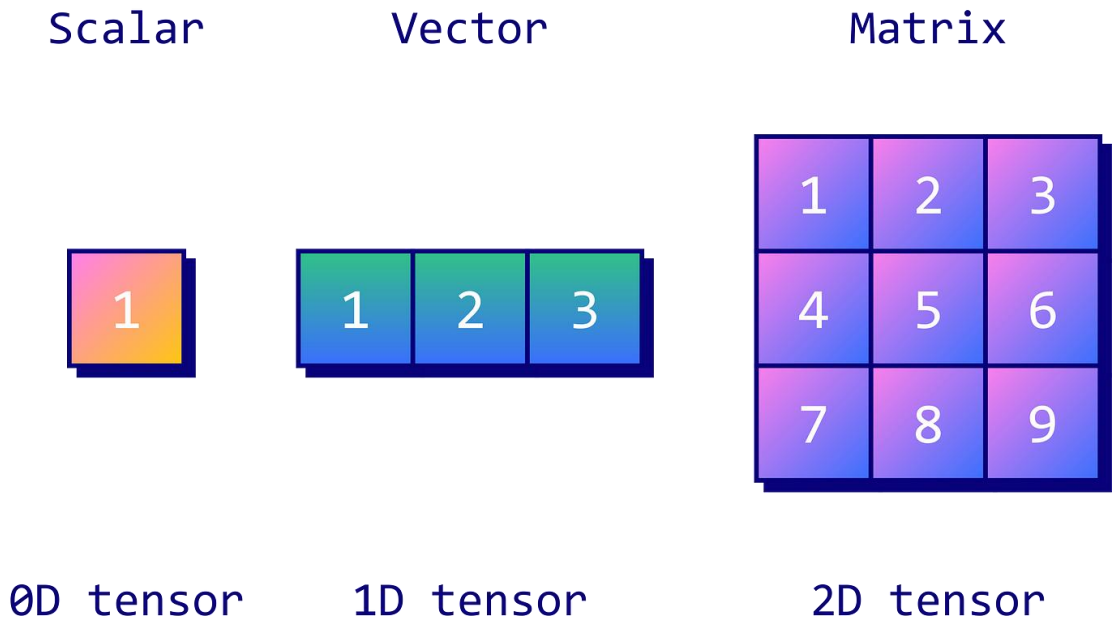
Guilherme Loan Schneider

Descrição da atividade

2. Deep Learning Fundamentals

O primeiro conteúdo utilizado nas aulas é o Tensor, que consiste no formato básico de dados utilizado no treinamento de redes neurais utilizando o PyTorch. Esses dados podem ser organizados da mesma forma que vetores, matrizes, valores únicos, float, double, strings, dentre outros.

Os Tensors representam os nossos dados, parâmetros dos modelos e as predições.



Podemos também realizar operações matemáticas como soma, subtração, multiplicação, divisão, transposição, dentre inúmeras operações, tais quais realizamos com matrizes, vetores, etc.

Na imagem abaixo, podemos verificar o formato de um Tensor qualquer. Note que a visualização é X, Y, Z, então, para o segundo exemplo da imagem, o Tensor possui 2 linhas e 3 colunas e 2 dimensões.

```
sample = torch.tensor([1, 2, 3])
sample.shape
```

```
torch.Size([3])
```

```
x = torch.tensor([[1, 2, 3], [4, 5, 6]])
x.shape
```

```
torch.Size([2, 3])
```

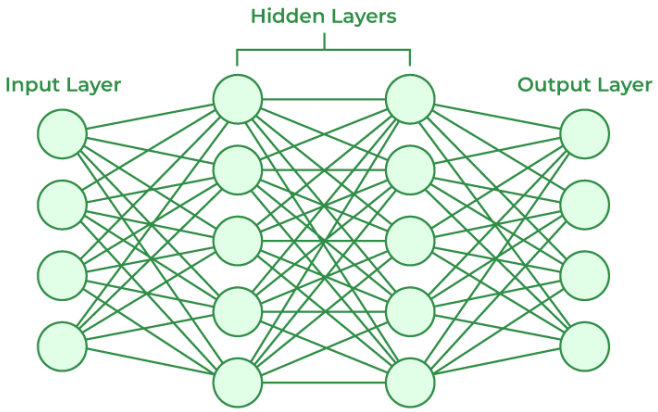
```
y = torch.tensor([[10], [11], [12]])
y.shape
```

```
torch.Size([3, 1])
```

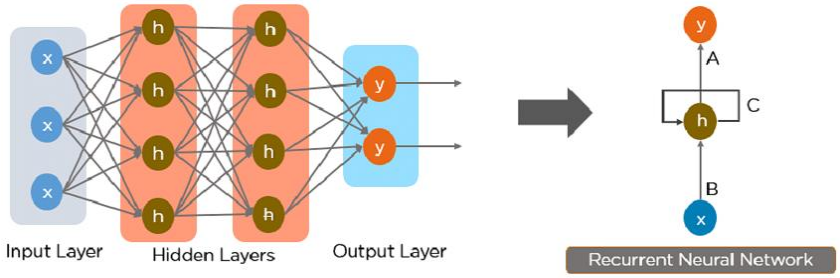
3. Building Neural Networks with PyTorch

Temos alguns tipos principais de redes neurais, ANNs, RNNs e CNNs, cada uma delas possui uma aplicação específica.

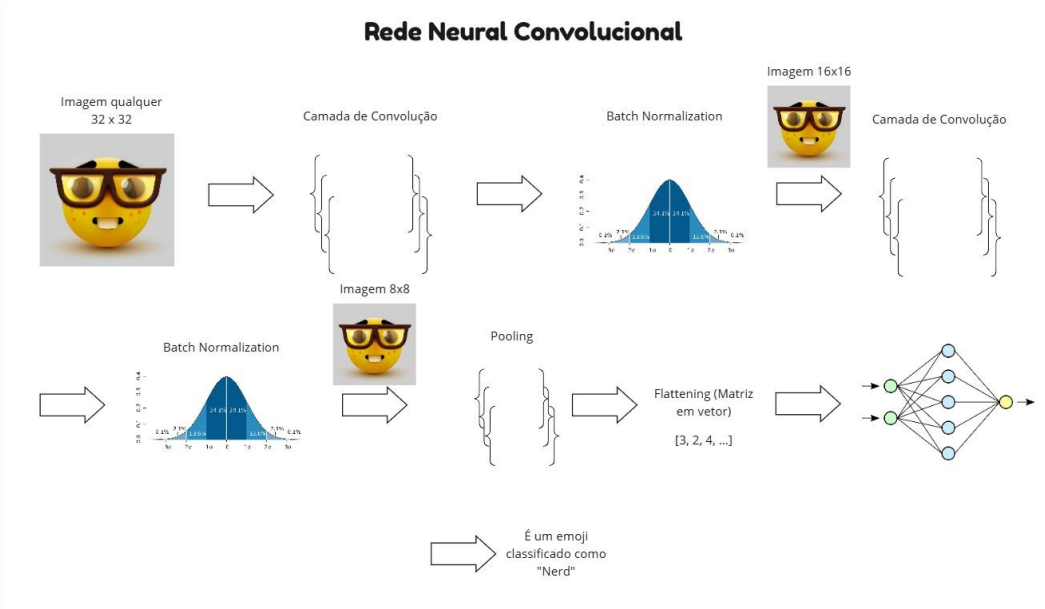
As Redes Neurais Artificiais (ANNs) são modelos computacionais inspirados no cérebro humano e são amplamente utilizadas para resolver problemas complexos que envolvem classificação, regressão e previsão. Elas funcionam bem em tarefas onde os dados são estruturados em formato tabular, como em diagnósticos médicos, previsões financeiras, e sistemas de recomendação.



As Redes Neurais Recorrentes (RNNs) são uma extensão das ANNs, projetadas especificamente para lidar com dados sequenciais, como séries temporais e linguagem natural. Diferenciam-se por manterem uma memória dos estados anteriores por meio de conexões recorrentes, o que as torna ideais para tarefas como tradução automática, geração de texto, análise de sentimentos e previsão de séries temporais.



As Redes Neurais Convolucionais (CNNs) são amplamente utilizadas em tarefas de processamento de imagens e vídeos, devido à sua capacidade de extrair características espaciais. Utilizando camadas de convolução, normalização e pooling, essas redes são capazes de identificar padrões como bordas, texturas e objetos, sendo fundamentais em aplicações como reconhecimento facial, classificação de imagens, diagnóstico por imagem na medicina e visão computacional em veículos autônomos.



Na construção de uma rede neural artificial, temos a seguinte estrutura para prever a soma de 2 valores em um vetor. Nesse caso temos o vetor de entrada, e de saída (o treinamento é supervisionado).

A primeira camada possui 2 neurônios de entrada e 8 neurônios na camada oculta. Em seguida temos a camada de ativação ReLU, e por fim a última camada que conecta os 8 neurônios na camada oculta com a de saída, que possui 1 neurônio apenas.

```
model = nn.Sequential(nn.Linear(2, 8),
                      nn.ReLU(),
                      nn.Linear(8, 1)).to(device='cuda' if torch.cuda.is_available() else 'cpu')
```

```
x = [[1,2], [3,4], [5,6], [7,8]]
y = [[3], [7], [11], [15]]
```

O autor utilizou o código abaixo para criar um “Dataset personalizado”, que permite com que utilizamos o DataLoader para treinamento dos modelos.

```
class MyDataSet(Dataset):
    def __init__(self, x, y):
        self.x = torch.tensor(x).float().to(device='cuda' if torch.cuda.is_available() else 'cpu')
        self.y = torch.tensor(y).float().to(device='cuda' if torch.cuda.is_available() else 'cpu')

    def __len__(self):
        return len(self.x)

    def __getitem__(self, idx):
        return self.x[idx], self.y[idx]
```

Em seguida utilizamos o DataLoader para carregar o dataset.

```
dl = DataLoader(ds, batch_size=2, shuffle=True)
```

Por fim, o treinamento é feito utilizando 1000 épocas. Os valores de erro são sempre armazenados no vetor loss_hist para posteriormente analisar o erro do modelo.

```
loss_fn = nn.MSELoss()
from torch.optim import SGD
optimizer = SGD(model.parameters(), lr=0.001)
import time
loss_hist = []
start = time.time()

model.train()
for _ in range(1000):
    for ix, iy in dl:
        optimizer.zero_grad()
        loss_value = loss_fn(model(ix), iy)
        loss_value.backward()
        optimizer.step()
        loss_hist.append(loss_value)
end = time.time()
print("Time taken: ", end - start)
```

Para testarmos o modelo, passamos um vetor teste com um vetor com pares de valores, visto que nosso modelo foi feito para realizar uma soma de apenas dois valores.

```
val = [[8,9], [10,11], [12,13], [14,15]]
model(torch.tensor(val).float().to(device='cuda' if torch.cuda.is_available() else 'cpu'))
```

A saída produzida está sendo mostrada na imagem abaixo, indicando que a primeira tupla resulta em 17, a segunda 21, a terceira 25 e a última 29.

```
tensor([[16.9979],
        [20.9965],
        [24.9951],
        [28.9937]], grad_fn=<AddmmBackward0>)
```

4. Neural Networks for Images

Nessa seção utilizamos o Dataset FMINIST para efetuar a classificação de imagens utilizando uma rede neural simples (note que não estamos utilizando CNNs). A

imagem abaixo demonstra algumas imagens contidas no banco de dados, que consistem em peças de vestuário tanto masculino quanto feminino.

Aqui nós fizemos o mesmo processo aplicado na seção anterior, adicionando uma etapa de flattening nas imagens do dataset para transformá-las em um vetor 1D, o parâmetro -1 torna o código mais “genérico” não precisando especificar o número de imagens.

```
class FMNISTDataset(Dataset):
    def __init__(self, x, y):
        x = x.float()
        x = x.view(-1, 28*28) # Aplicando o flattening para cada imagem
        self.x, self.y = x, y

    def __len__(self):
        return len(self.x)

    def __getitem__(self, idx):
        x,y = self.x[idx], self.y[idx]
        return x.to(device), y.to(device)
```

Em seguida a rede neural criada é especificada pela imagem abaixo, sendo muito similar a anterior, até mesmo muito simples para um problema de classificação de imagem.

```
from torch.optim import SGD
def get_model():
    model = nn.Sequential(
                nn.Linear(28*28, 1000), # Flattening
                nn.ReLU(),
                nn.Linear(1000, 10)
            ).to(device)
    loss_fn = nn.CrossEntropyLoss()
    optimizer = SGD(model.parameters(), lr=1e-2)
    return model, loss_fn, optimizer
```

Com as funções abaixo poderemos fazer a avaliação do modelo, juntamente com o código de treinamento para cada batch.

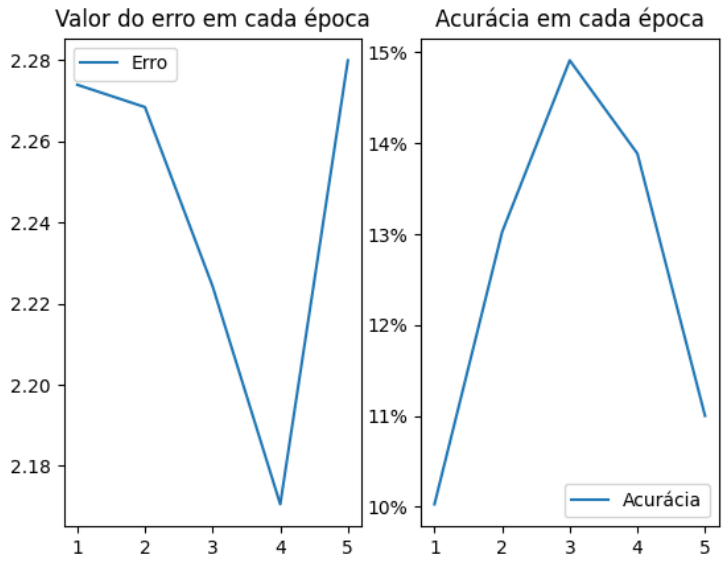
```
@torch.no_grad()
def accuracy(x,y, model):
    model.eval()
    predictions = model(x)
    max_values, argmaxes = predictions.max(-1)
    is_correct = argmaxes == y
    acc = is_correct.numpy().mean()
    return acc.item()
```

```
def train_branch(x,y,model,opt,loss_fn):
    model.train()
    predictions = model(x)
    loss = loss_fn(predictions, y)
    loss.backward()
    opt.step()
    opt.zero_grad()
    return loss.item()
```

Por fim, é feito a execução do treinamento do modelo, em cinco épocas. Novamente, os valores de erro e acurácia por época são armazenados em vetores.

```python
losses, accs = [], []
for epoch in range(5):
    print(epoch)
    epoch_losses, epoch_accs = [], []
    for ix, batch in enumerate(iter(trn_dl)):
        x, y = batch
        loss = train_branch(x,y,model,opt,loss_fn)
        epoch_losses.append(loss)
    epoch_losses = np.array(epoch_losses).mean()
    for ix, batch in enumerate(iter(trn_dl)):
        x, y = batch
        is_correct = accuracy(x,y,model)
        epoch_accs.append(is_correct)
    epoch_accs = np.mean(epoch_accs)
    losses.append(epoch_losses)
    accs.append(epoch_accs)
```

Na imagem abaixo, podemos analisar que o resultado do modelo é precário para esse dataset, o que indica que o modelo não está conseguindo distinguir as imagens. Muito provavelmente isso se dá por conta do modelo ser muito simples, não possuindo parâmetros suficientes para distinguir 10 classes diferentes.



5. Convolutional Neural Networks (CNNs)

Nessa seção, o enfoque foi nas CNNs voltadas para o problema da seção anterior, o FMNIST. Em primeiro plano, o autor definiu um Data Augmentation a fim de melhorar os resultados do algoritmo.

```python
import albumentations as A

aug = A.Compose([
    A.Affine(translate_px={'x': (-10, 10), 'y': (0, 0)}, mode=0, p=1.0)
])
```

Os valores de ‘x’ indicam que a imagem pode ser deslocada horizontalmente entre -10 e 10 pixels, já 'y' indica que não haverá deslocamento vertical.

Na seção abaixo temos códigos importantes que irão aplicar o data augmentation no dataset, juntamente com o agrupamento desses dados em lotes, isso será feito pela função collate_fn no momento em que a FMNISTDataset for chamada. Além disso, as imagens serão automaticamente convertidas para tensores e normalizadas

```
class FMNISTDataset(Dataset):
    def __init__(self, x, y, aug=None):
        self.x, self.y = x, y
        self.aug = aug

    def __getitem__(self, ix):
        x, y = self.x[ix], self.y[ix]
        return x, y

    def __len__(self):
        return len(self.x)

    def collate_fn(self, batch):
        ims, classes = list(zip(*batch))
        ims_np = np.stack([im.numpy() for im in ims])
        if self.aug:
            ims_aug = []
            for im in ims_np:
                im = np.expand_dims(im, axis=-1)
                augmented = self.aug(image=im)
                ims_aug.append(augmented['image'])
            ims_np = np.stack(ims_aug)
            ims_np = ims_np.squeeze(-1)
        ims = torch.tensor(ims_np)[:, None, :, :].float().to(device) / 255.
        classes = torch.tensor(classes).to(device)
        return ims, classes
```

Na imagem abaixo, a CNN é definida, recebendo uma imagem de entrada, essa por sua vez passará por etapas de convolução com kernel de tamanho 3 e camadas de MaxPooling. Em seguida, a imagem é transformada em um vetor 1D pela camada de flatten e em seguida começa a combinar as características extraídas para classificação. Na última camada a classificação é feita entre as 10 saídas possíveis.

```
from torch.optim import SGD, Adam
def get_model():
    model = nn.Sequential(
        nn.Conv2d(1, 64, kernel_size=3),
        nn.MaxPool2d(2),
        nn.Conv2d(64, 128, kernel_size=3),
        nn.MaxPool2d(2),
        nn.ReLU(),
        nn.Flatten(),
        nn.Linear(3200, 256),
        nn.ReLU(),
        nn.Linear(256, 10)
    ).to(device)
    loss_fn = nn.CrossEntropyLoss()
    optimizer = Adam(model.parameters(), lr=1e-3)
    return model, loss_fn, optimizer
```

Em seguida definimos as funções de treino e de carregar os dados. Note o uso do collate_fn definido anteriormente na montagem do FMNISTDataset.

```python
def train_batch(x, y, model, opt, loss_fn):
    model.train()
    prediction = model(x)
    batch_loss = loss_fn(prediction, y)
    batch_loss.backward()
    opt.zero_grad()
    opt.step()
    return batch_loss.item()

def get_data():
    train = FMNISTDataset(tr_images, tr_targets, aug=aug)
    trn_dl = DataLoader(train, batch_size=64,
                        collate_fn=train.collate_fn, shuffle=True)
    val = FMNISTDataset(val_images, val_targets)
    val_dl = DataLoader(val, batch_size=len(val_images),
                        collate_fn=val.collate_fn, shuffle=True)
    return trn_dl, val_dl
```

Em seguida o modelo é treinado por 5 épocas, armazenando a loss do treinamento.

```python
trn_dl, val_dl = get_data()
model, loss_fn, optimizer = get_model()
for epoch in range(5):
    print(f"Epoch {epoch}")  # Exibe a época atual
    for ix, batch in enumerate(iter(trn_dl)):
        x, y = batch  # Obtém o lote de dados
        batch_loss = train_batch(x, y, model, optimizer, loss_fn)  # Treina o modelo e calcula a perda
```

Conseguimos analisar que o desempenho do algoritmo continua precário, com altíssima loss no conjunto de validação e acurácia de 11.75% no mesmo conjunto.

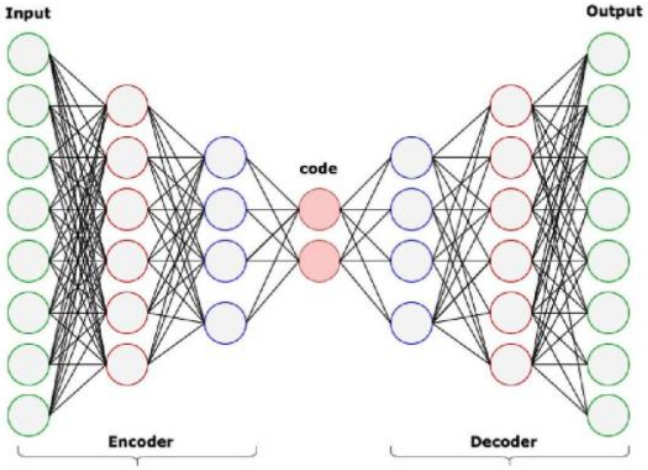
```python
# Avaliando o modelo
model.eval()
with torch.no_grad():
    for ix, batch in enumerate(val_dl):
        x, y = batch
        prediction = model(x)  # Faz a previsão
        val_loss = loss_fn(prediction, y)  # Calcula a perda de validação
        print(f"Validation Loss: {val_loss.item()}")  # Exibe a perda de validação

# Analisando a acurácia do modelo
correct = 0
total = 0
with torch.no_grad():
    for ix, batch in enumerate(val_dl):
        x, y = batch
        prediction = model(x)  # Faz a previsão
        _, predicted = torch.max(prediction.data, 1)  # Obtém as classes previstas
        total += y.size(0)  # Total de amostras
        correct += (predicted == y).sum().item()  # Contagem de acertos
accuracy = 100 * correct / total
print(f'Acurácia do modelo nos dados de validação: {accuracy:.2f}%')
```

```
Validation Loss: 2.3036913871765137
Acurácia do modelo nos dados de validação: 11.75%
```

6. Auto Encoders

As redes neurais AutoEncoders são um tipo específico treinado para copiar sua entrada para sua saída. A ideia principal desse tipo de rede é aprender uma representação codificada dos dados, chamada de "representação latente", reduzindo a dimensionalidade da entrada no processo. Isso é feito por meio de duas partes principais: o codificador (encoder), que comprime os dados em uma forma reduzida, e o decodificador (decoder), que tenta reconstruir a entrada original a partir dessa representação comprimida.



Autoencoders são úteis para redução de dimensionalidade, remoção de ruído (denoising autoencoders), geração de dados e aprendizado não supervisionado de características. Eles são treinados usando um critério de erro de reconstrução, como o erro quadrático médio entre a entrada e a saída reconstruída, incentivando a rede a aprender os padrões mais relevantes dos dados.

Nessa seção vale ressaltar a diferença na construção das redes neurais, onde existem duas partes principais, o encoder e o decoder.

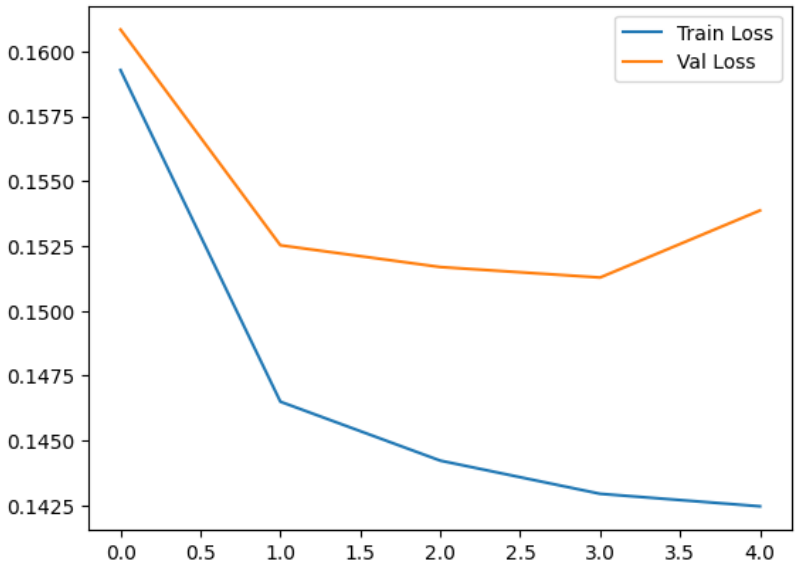
Em vermelho temos a parte do encoder, que mapeia uma imagem 28x28 para um espaço latente de dimensão definida pelo parâmetro “latent_dim”. Essa parte da rede é responsável por capturar e comprimir as principais características da imagem em uma representação mais compacta.

Já em amarelo está o decoder, cuja função é reconstruir a imagem original a partir do vetor latente gerado pelo encoder. Ele faz isso aprendendo a reverter a codificação, projetando o vetor latente de volta para o espaço original de dimensão 28x28, produzindo uma saída que idealmente seja o mais próxima possível da entrada original.

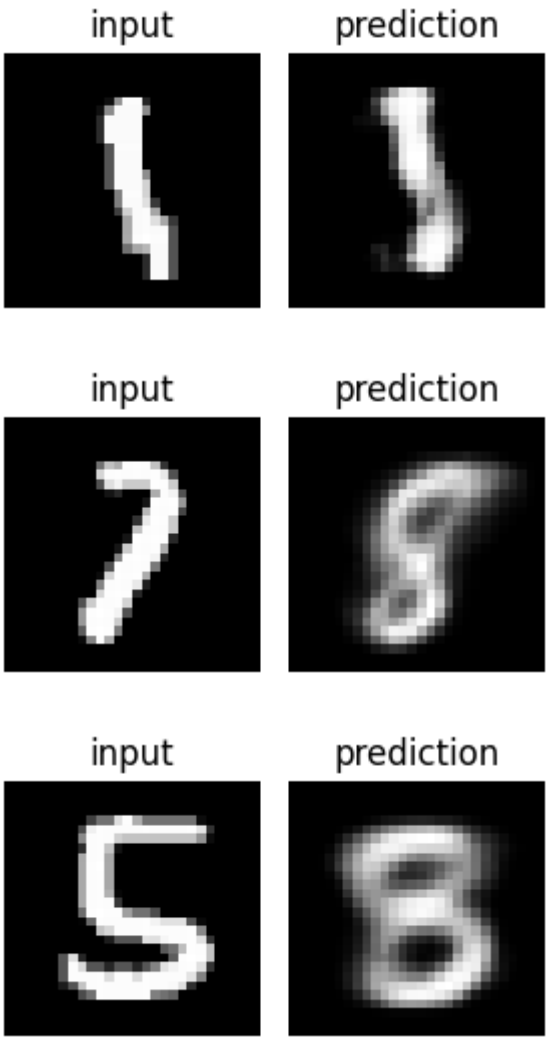
```python
class AutoEncoder(nn.Module):
    def __init__(self, latent_dim):
        super().__init__()
        self.latent_dim = latent_dim
        self.encoder = nn.Sequential(
            nn.Linear(28*28,128), nn.ReLU(True),
            nn.Linear(128,64), nn.ReLU(True),
            nn.Linear(64,latent_dim))
        self.decoder = nn.Sequential(
            nn.Linear(latent_dim,64),nn.ReLU(True),
            nn.Linear(64,128),nn.ReLU(True),
            nn.Linear(128,28*28),nn.Tanh())

    def forward(self,x):
        x = x.view(len(x), -1)
        x = self.encoder(x)
        x = self.decoder(x)
        x = x.view(len(x), 1, 28, 28)
        return x
```

Utilizando 5 épocas no treinamento desse modelo e um vetor latente de dimensão igual a 3, obtivemos o resultado abaixo, com erro no treinamento de 0.1425 e nos dados de validação de 0.1539.



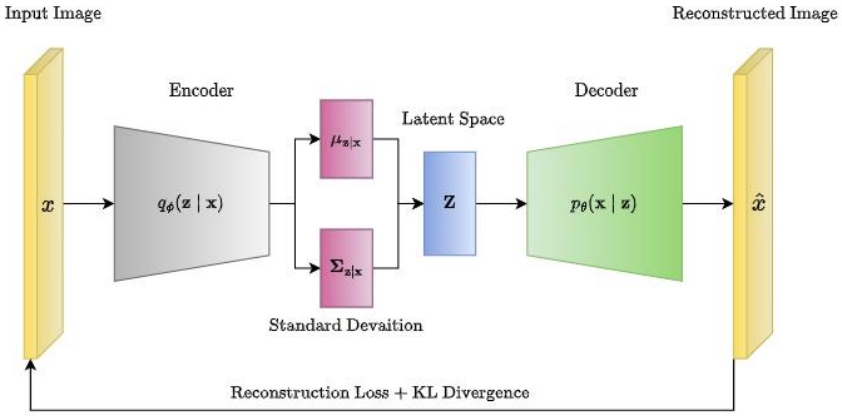
Analisando a imagem abaixo, que possui as outputs produzidas por essa rede neural autoencoder, podemos verificar que algumas predições estão incorretas, como é o caso da 2ª e 3ª imagem, mas a reconstrução feita pelo algoritmo é muito fiel a realidade, possuindo apenas um Blur.



Variational Autoencoder

VAE, ou Autoencoder Variacional, é um tipo de modelo de aprendizado profundo usado principalmente para aprendizado de representação e geração de dados. Ele se destaca por aprender uma representação latente contínua dos dados, permitindo a geração de novas amostras similares às originais.

A diferença fundamental dos VAEs é que, em vez de aprender um único ponto no espaço latente para cada entrada, eles aprendem uma distribuição de probabilidade (a partir da inferência bayesiana). Isso permite que o modelo gere novas amostras aleatoriamente a partir dessa distribuição.



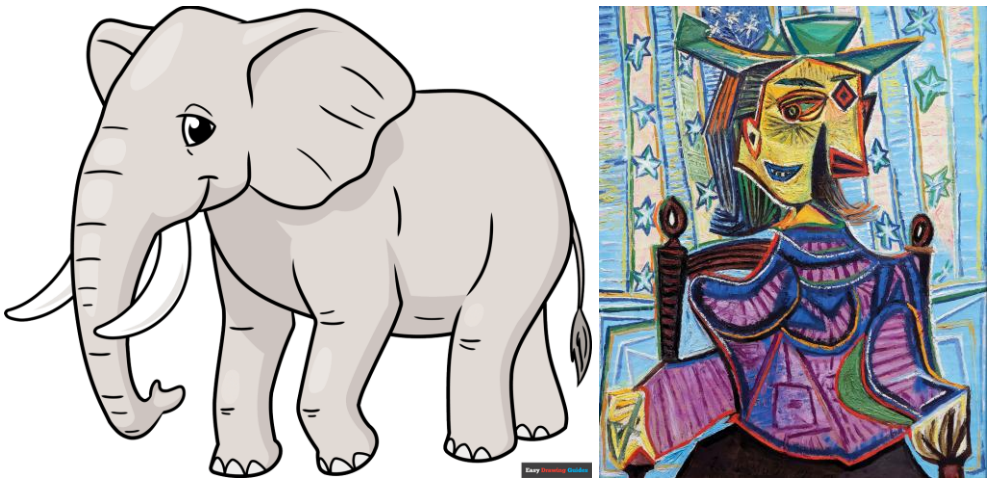
7. Hands-on Projects

Nessa seção, o conteúdo abordado é mais voltado para projetos práticos, que abordaram a geração de rostos utilizando VAEs (essa prática não foi possível ser feita por conta de que o dataset com rostos foi excluído), colorização automática de imagens com autoencoders e por fim a mesclagem de estilos entre duas imagens.

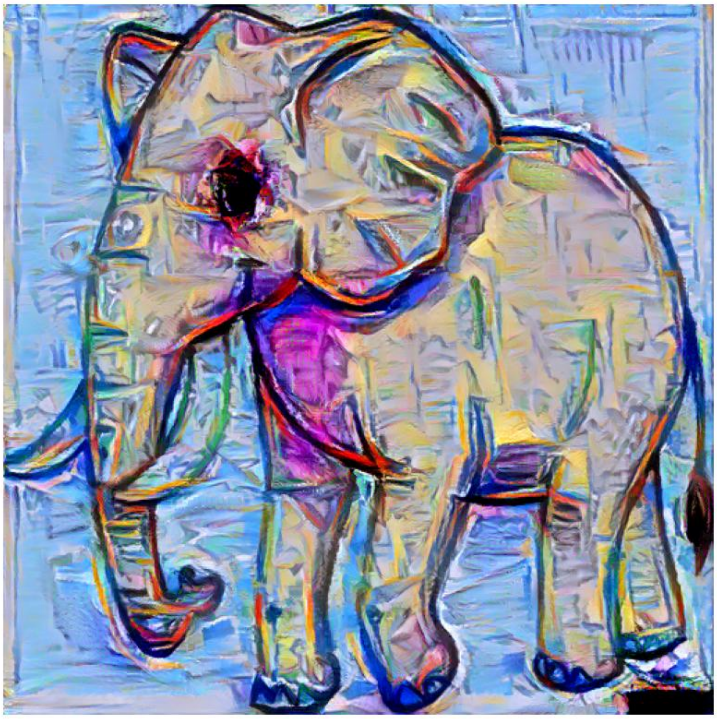
A Figura abaixo demonstra a colorização automática, onde:

1. A imagem mais à esquerda: original em preto e branco (entrada)
2. A imagem central: original colorida (saída esperada);
3. A imagem mais à direita: reconstrução pelo modelo (saída do modelo).

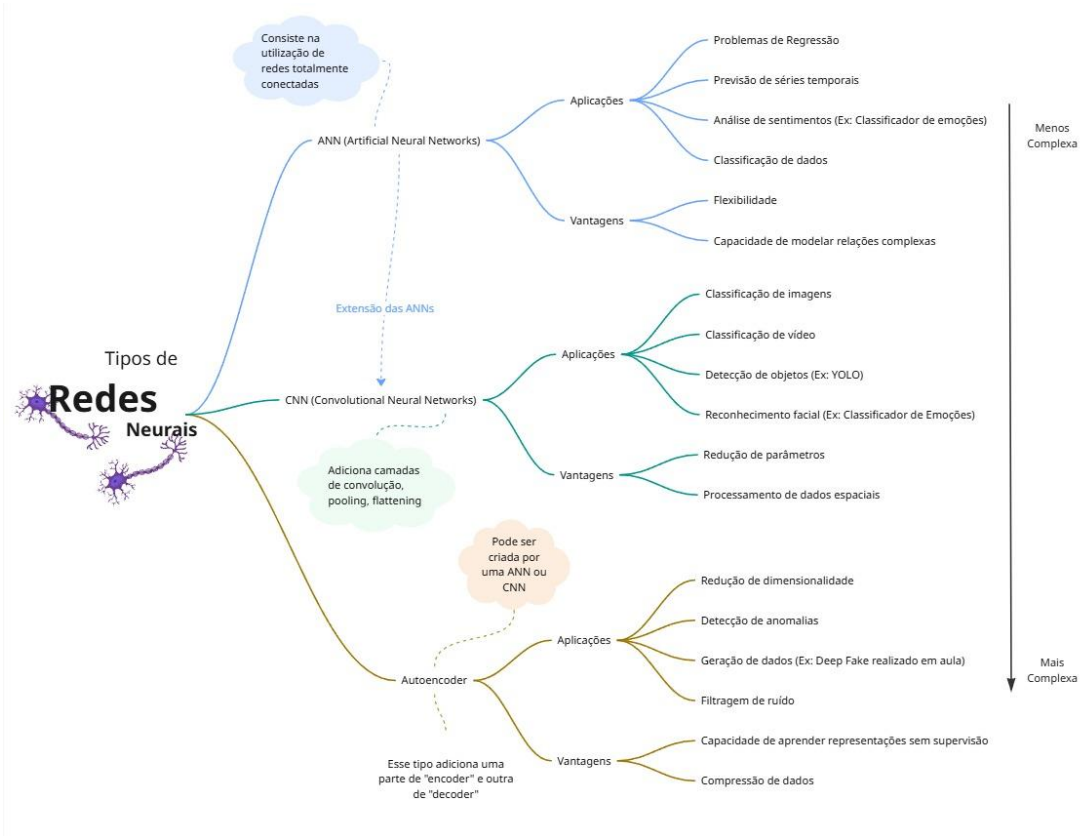
Em seguida, no projeto de mesclagem de estilos, se deu pelas duas imagens abaixo, onde o estilo da figura à direita será aplicado na imagem à esquerda. Nesse processo utiliza-se uma rede neural VGG19 modificada.



A saída do modelo foi a seguinte:



Insight sobre os tipos de redes neurais e suas principais aplicações/vantagens.



Referencias

The Ultimate Computer Vision and Deep Learning Course (Seção 1 a 7)