Centro de Informática - UFPE

Introdução à Aprendizagem Profunda

Aluno: Victor Gabriel de Carvalho (vgc3)

Professor: Tsang Ing Ren / George Darmiton da Cunha Cavalcanti

LISTA PRÁTICA DAS UNIDADES 1 E 2

Pode ser feita com o grupo do projeto. Recomendo pair/group programming para que todos vejam um pouco de todas as partes.

Treine e avalie 4 modelos de classificação para a base de dados do FashionMNIST (https://www.kaggle.com/datasets/zalando-research/fashionmnist, https://pytorch.org/vision/stable/generated/torchvision.datasets.FashionMNIST.html).

1. Um modelo base que não seja uma rede neural, como *decision tree, xgboost, random forest*, etc. Recomendação: use o sklearn (https://scikit-learn.org/).
2. Uma MLP
3. Uma rede convolucional criada por ti. Recomendação: https://pytorch.org/
4. Use um modelo pré treinado já consolidado na literatura para fazer *transfer learning*. Recomendações: https://pytorch.org/hub/pytorch_vision_vgg/

Compare os resultados dos modelos:

- plote gráficos que mostrem as acurácias de cada modelo
- Indique qual foi a classe na qual o modelo teve pior performance (indique qual métrica usou para concluir isso e faça para cada modelo)
- argumente qual o melhor modelo levando em consideração o tempo de execução e acurácia.

Recomendação use:
https://pytorch.org/vision/main/generated/torchvision.datasets.MNIST.html .

Recomendação:

Faça um template de treino, validação e teste que funcione para uma API de modelo.

Crie a API para cada modelo que será usado e use o template

Imports e Downloads

```
import time
import psutil
import numpy as np

import torch
from torch import nn
from torch.utils.data import DataLoader, SubsetRandomSampler
from torchvision.datasets import FashionMNIST
from torchvision.transforms import v2
from torchvision.models import vgg16, VGG16_Weights, resnet34,
ResNet34_Weights, mobilenet_v3_small, MobileNet_V3_Small_Weights

import matplotlib.pyplot as plt
import seaborn as sns

from tqdm.auto import tqdm

from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier,
GradientBoostingClassifier
from sklearn.svm import SVC
from sklearn.ensemble import AdaBoostClassifier

from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report, accuracy_score,
confusion_matrix
```

Definiremos uma random seed para garantir reprodutibilidade. Nos métodos do `sklearn`(usados pelo modelo base), a random seed é passada como parâmetro.

```
torch.manual_seed(9)
torch.cuda.manual_seed(9)

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
device
```

```
device(type='cuda')
```

Dataset

O trecho de código abaixo faz o download do dataset Fashion MNIST, que já está incluso no `torchvision.datasets`, eliminando a necessidade de downloads adicionais. O dataset é composto por 60000 imagens de treinamento e 10000 imagens de teste. As imagens tem a resolução de (28 x 28) e são em escala de cinza (1 x 28 x 28).

> Usaremos a classe customizada `MyDataset` para podermos aplicar diferentes transformações ao longo das implementações.

```
train_dataset = FashionMNIST(root='./data', train=True, download=True)
test_dataset = FashionMNIST(root='./data', train=False, download=True)

Downloading http://fashion-mnist.s3-website.eu-central-
1.amazonaws.com/train-images-idx3-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-
1.amazonaws.com/train-images-idx3-ubyte.gz to
./data/FashionMNIST/raw/train-images-idx3-ubyte.gz

100%|██████████| 26421880/26421880 [00:01<00:00, 14949730.71it/s]

Extracting ./data/FashionMNIST/raw/train-images-idx3-ubyte.gz to
./data/FashionMNIST/raw

Downloading http://fashion-mnist.s3-website.eu-central-
1.amazonaws.com/train-labels-idx1-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-
1.amazonaws.com/train-labels-idx1-ubyte.gz to
./data/FashionMNIST/raw/train-labels-idx1-ubyte.gz

100%|██████████| 29515/29515 [00:00<00:00, 267680.82it/s]

Extracting ./data/FashionMNIST/raw/train-labels-idx1-ubyte.gz to
./data/FashionMNIST/raw

Downloading http://fashion-mnist.s3-website.eu-central-
1.amazonaws.com/t10k-images-idx3-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-
1.amazonaws.com/t10k-images-idx3-ubyte.gz to
./data/FashionMNIST/raw/t10k-images-idx3-ubyte.gz

100%|██████████| 4422102/4422102 [00:00<00:00, 4863281.18it/s]

Extracting ./data/FashionMNIST/raw/t10k-images-idx3-ubyte.gz to
./data/FashionMNIST/raw

Downloading http://fashion-mnist.s3-website.eu-central-
1.amazonaws.com/t10k-labels-idx1-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-
1.amazonaws.com/t10k-labels-idx1-ubyte.gz to
./data/FashionMNIST/raw/t10k-labels-idx1-ubyte.gz

100%|██████████| 5148/5148 [00:00<00:00, 2254361.77it/s]

Extracting ./data/FashionMNIST/raw/t10k-labels-idx1-ubyte.gz to
./data/FashionMNIST/raw


class MyDataset(torch.utils.data.Dataset):
    def __init__(self, dataset, transform=None):
        self.dataset = dataset
        self.transform = transform
```

```
    def __getitem__(self, index):
        x, y = self.dataset[index]
        if self.transform:
            x = self.transform(x)
        return x, y

    def __len__(self):
        return len(self.dataset)
```

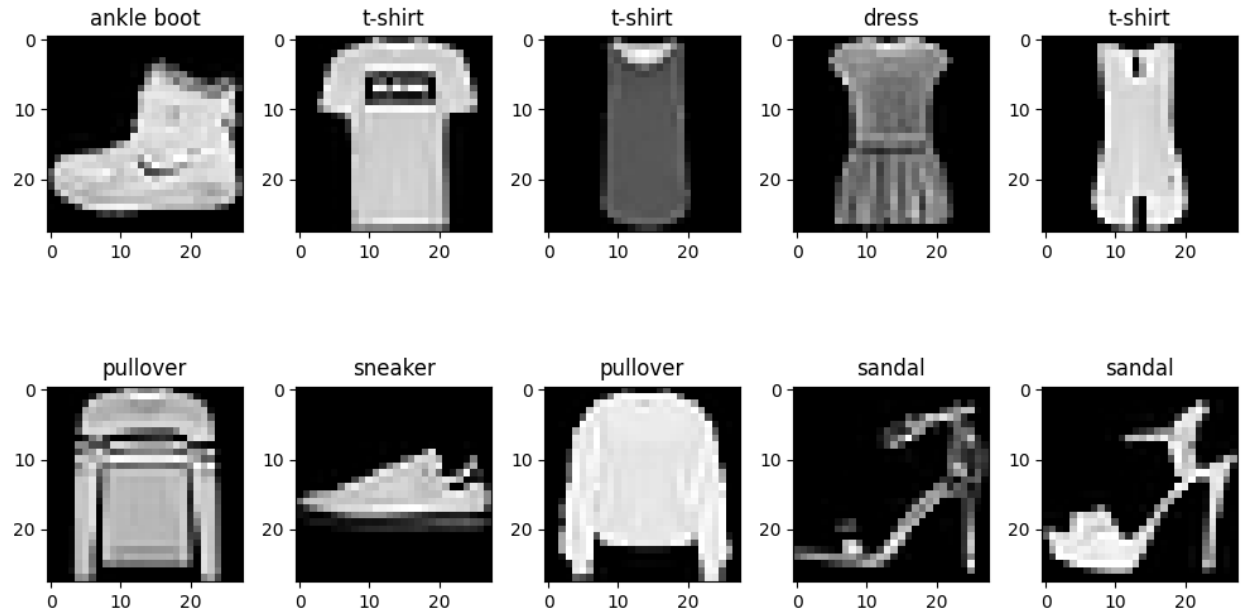
Dataset para o Modelo Base

A versão do dataset usada no modelo base consistirá apenas do conjunto de treinamento e teste e não será necessário um dataloader. Iremos apenas dividir o `train_dataset` e `test_dataset` entre as features (imagem) `X_` e as labels (classes) `y_`.

```
base_transforms = v2.Compose([
    v2.ToImage(), v2.ToDtype(torch.float32, scale=True)
])

train_base_dataset = MyDataset(train_dataset, base_transforms)
test_base_dataset = MyDataset(test_dataset, base_transforms)

X_train = np.array([np.array(sample[0]) for sample in
train_base_dataset])
y_train = np.array([np.array(sample[1]) for sample in
train_base_dataset])

X_test = np.array([np.array(sample[0]) for sample in
test_base_dataset])
y_test = np.array([np.array(sample[1]) for sample in
test_base_dataset])

X_train.shape, y_train.shape, X_test.shape, y_test.shape
```

```
((60000, 1, 28, 28), (60000,), (10000, 1, 28, 28), (10000,))
```

```
labels_title = ['t-shirt', 'trouser', 'pullover', 'dress', 'coat',
'sandal', 'shirt', 'sneaker', 'bag', 'ankle boot']

fig, ax = plt.subplots(2, 5, figsize=(10, 6))
for idx in range(10):
    ax[idx//5][idx%5].imshow(X_train[idx].reshape(28, 28),
cmap='gray')
    ax[idx//5][idx%5].set_title(labels_title[y_train[idx]])
plt.tight_layout()
plt.show()
```

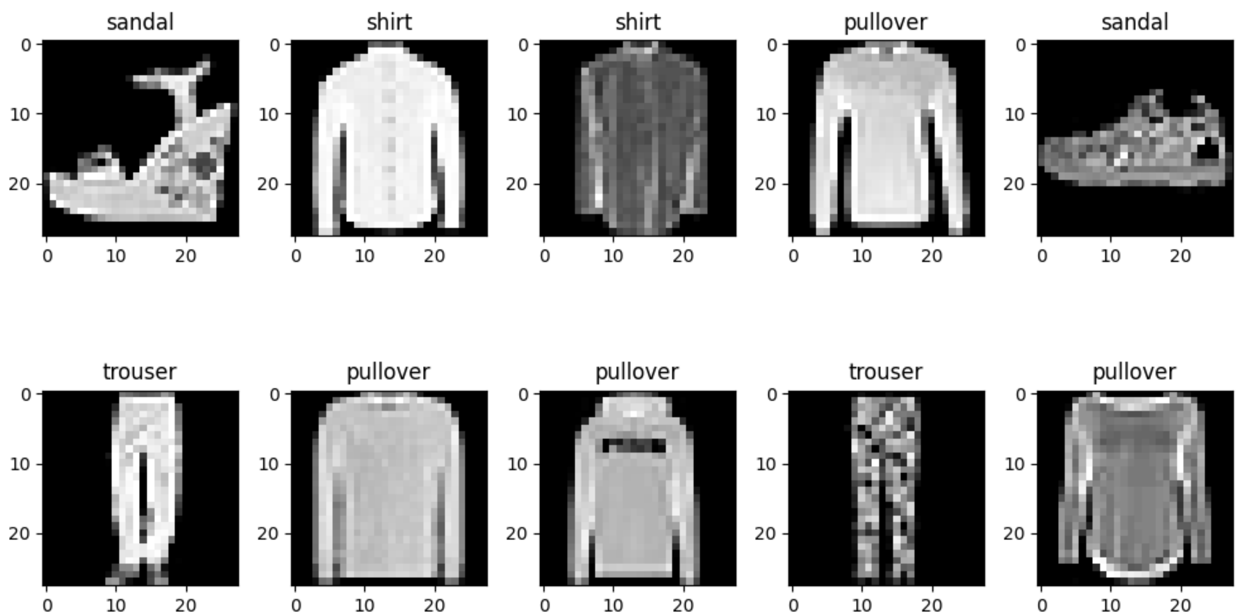


Dataset para MLP e CNN

Nesse caso, devido à maior flexibilidade do PyTorch, podemos dividir os dados em treinamento, validação e teste, além de possibilitar o uso de dataloaders para carregarmos bathces de dados (possiblitando, também, a data). O trecho de código abaixo faz a divisão dos datasets, onde 20% dos dados de treinamento irão para a validação.

```
mlp_cnn_transforms = v2.Compose([
    v2.RandomHorizontalFlip(),
    v2.ColorJitter(brightness=0.5, contrast=0.5, saturation=0.5,
hue=0.5),
    v2.ToImage(), v2.ToDtype(torch.float32, scale=True)
])

train_mlp_cnn_dataset = MyDataset(train_dataset, mlp_cnn_transforms)
test_mlp_cnn_dataset = MyDataset(test_dataset, base_transforms)

num_samples = len(train_dataset)
indices = list(range(num_samples))

split_ratio = 0.8
split_idx = int(num_samples * split_ratio)

train_indices, val_indices = indices[:split_idx], indices[split_idx:]
train_sampler = SubsetRandomSampler(train_indices)
val_sampler = SubsetRandomSampler(val_indices)

BATCH_SIZE = 128
train_mlp_cnn_loader = DataLoader(train_mlp_cnn_dataset,
batch_size=BATCH_SIZE, sampler=train_sampler)
val_mlp_cnn_loader = DataLoader(train_mlp_cnn_dataset,
```

```
batch_size=BATCH_SIZE, sampler=val_sampler)
test_mlp_cnn_loader = DataLoader(test_mlp_cnn_dataset,
batch_size=BATCH_SIZE, shuffle=True)

it = iter(train_mlp_cnn_loader)
images, labels = next(it)

fig, ax = plt.subplots(2, 5, figsize=(10, 6))
for idx in range(10):
    ax[idx//5][idx%5].imshow(images[idx].permute(1, 2, 0),
cmap='gray')
    ax[idx//5][idx%5].set_title(labels_title[labels[idx]])
plt.tight_layout()
plt.show()
```



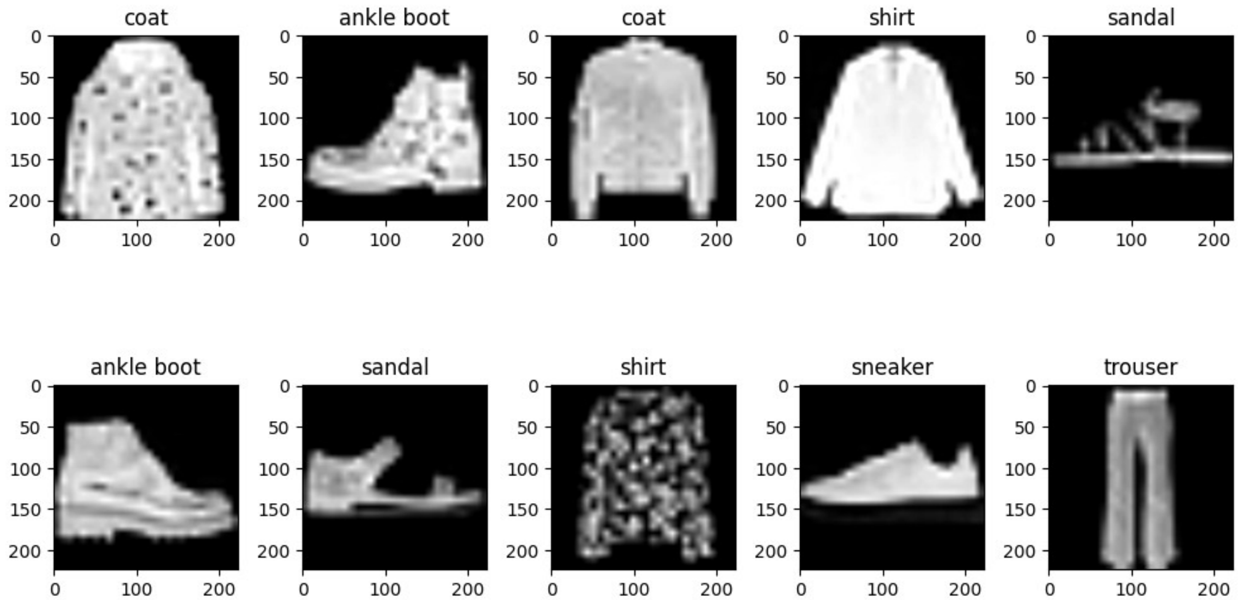
Dataset para CNNs consolidadas

A maioria das CNNs consolidadadas foi projetada para trabalhar com o dataset ImageNet, que contem imagens (3 x 224 x 224). Dessa forma, as imagens bem menores do FashionMNIST não conseguiriam ser processadas por essas redes (seu tamanho desapareceria após uma certa quantidade de camadas pooling).

Pare resolver esse problema e mantermos fiel às arquiteturas originais das CNNs, iremos apenas das um `v2.Resize` nas imagens do dataset (mesmo sabendo que tal operação não adicionaria informação nenhuma, apenas gastaria mais recursos). Podemos perceber o reajuste através dos eixos do `plt.imshow`.

```
cnn_cons_transforms = v2.Compose([
    v2.Resize((224, 224), antialias=True),
    v2.RandomHorizontalFlip(),
```

```
    v2.ColorJitter(brightness=0.5, contrast=0.5, saturation=0.5,
hue=0.5),
    v2.ToImage(), v2.ToDtype(torch.float32, scale=True)
])

cnn_cons_data_transforms = v2.Compose([
    v2.Resize((224, 224), antialias=True),
    v2.ToImage(), v2.ToDtype(torch.float32, scale=True)
])

train_cnn_cons_dataset = MyDataset(train_dataset, cnn_cons_transforms)
test_cnn_cons_dataset = MyDataset(test_dataset,
cnn_cons_data_transforms)

BATCH_SIZE = 64
train_cnn_cons_loader = DataLoader(train_cnn_cons_dataset,
batch_size=BATCH_SIZE, sampler=train_sampler)
val_cnn_cons_loader = DataLoader(train_cnn_cons_dataset,
batch_size=BATCH_SIZE, sampler=val_sampler)
test_cnn_cons_loader = DataLoader(test_cnn_cons_dataset,
batch_size=BATCH_SIZE, shuffle=True)

it = iter(train_cnn_cons_loader)
images, labels = next(it)

fig, ax = plt.subplots(2, 5, figsize=(10, 6))
for idx in range(10):
    ax[idx//5][idx%5].imshow(images[idx].permute(1, 2, 0),
cmap='gray')
    ax[idx//5][idx%5].set_title(labels_title[labels[idx]])
plt.tight_layout()
plt.show()
```



Funções Auxiliares

```
def train(model, train_loader, val_loader, max_epochs, loss_fn, optimizer, patience=5):
    train_loss_list = []
    val_loss_list = []
    val_acc_list = []

    best_val_loss = float('inf')
    counter = 0

    ram_usage = 0
    vram_allocated = 0
    time_0 = time.time()
    for epoch in range(1, max_epochs+1):
        train_loss = 0.0
        for (images, labels) in tqdm(train_loader):
            images, labels = images.to(device), labels.to(device)

            model.train()
            y_pred = model(images)

            loss = loss_fn(y_pred, labels)
            train_loss += loss.item()

            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

            ram_usage = psutil.virtual_memory().used / (1024 ** 3)
            vram_allocated = torch.cuda.max_memory_allocated(device) / (1024 ** 3)
```

```
        val_acc = 0.0
        val_loss = 0.0
        model.eval()
        with torch.inference_mode():
            for (images, labels) in tqdm(val_loader):
                images, labels = images.to(device), labels.to(device)

                y_val_pred = model(images)
                val_loss += loss_fn(y_val_pred, labels).item()
                val_acc += accuracy_score(labels.cpu(),
torch.argmax(y_val_pred, dim=1).cpu())

            val_loss /= len(val_loader)
            val_loss_list.append(val_loss)
            val_acc /= len(val_loader)
            val_acc_list.append(val_acc)

        train_loss /= len(train_loader)
        train_loss_list.append(train_loss)
        print(f"{epoch:02d}: Train loss: {train_loss:.5f}, RAM Usage:
{ram_usage:.2}GB, VRAM Usage: {vram_allocated:.2}GB | Validation loss:
{val_loss:.5f}, Validation acc: {(val_acc * 100):.2f}%")

        if val_loss < best_val_loss:
            best_val_loss = val_loss
            counter = 0
        else:
            counter += 1
            if counter >= patience:
                print(f"Early stopping at epoch {epoch}")
                break

    time_f = time.time()

    fig, ax = plt.subplots(figsize=(10,6))

    ax.plot(train_loss_list, label='Train loss')
    ax.plot(val_loss_list, label='Validation loss:')
    ax.set_title("Loss value during training")
    ax.set_xlabel('Epochs')
    ax.set_ylabel('Loss (CrossEntropy)')
    ax.legend()

    torch.cuda.reset_peak_memory_stats(device)
    return (time_f - time_0), ram_usage, vram_allocated


def test(model, test_loader, loss_fn):
    test_loss = 0.0
```

```python
    y_pred = []
    y_true = []

    time_0 = time.time()
    model.eval()
    with torch.inference_mode():
        for (images, labels) in tqdm(test_loader):
            images, labels = images.to(device), labels.to(device)

            y_test_pred_logits = model(images)
            test_loss += loss_fn(y_test_pred_logits, labels).item()

            y_test_pred = model.predict(images)
            y_pred.extend(y_test_pred.cpu().numpy())
            y_true.extend(labels.cpu().numpy())

        test_loss /= len(test_loader)

    time_f = time.time()
    return y_pred, y_true, test_loss, (time_f - time_0)
```

Nesse dicionário serão guardadas as informações de cada modelo. Esses dados serão usados em uma posterior análise.

Tr Time: Tempo de treinamento no dataset de treino.

Pr Time: Pempo de previsão no dataset de teste.

Accuracy: Acurácia no no dataset de teste.

RAM: Uso de RAM pelo sistema na ultima época.

VRAM: Uso máximo de VRAM da GPU.

```python
models = {'Decision Tree': {'Tr Time': 0, 'Pr Time': 0, 'Accuracy': 0, 'RAM': 0, 'VRAM': 0},
          'Random Forest': {'Tr Time': 0, 'Pr Time': 0, 'Accuracy': 0, 'RAM': 0, 'VRAM': 0},
          'SVM': {'Tr Time': 0, 'Pr Time': 0, 'Accuracy': 0, 'RAM': 0, 'VRAM': 0},
          'MLP': {'Tr Time': 0, 'Pr Time': 0, 'Accuracy': 0, 'RAM': 0, 'VRAM': 0},
          'Custom CNN': {'Tr Time': 0, 'Pr Time': 0, 'Accuracy': 0, 'RAM': 0, 'VRAM': 0},
          'VGG16': {'Tr Time': 0, 'Pr Time': 0, 'Accuracy': 0, 'RAM': 0, 'VRAM': 0},
          'ResNet34': {'Tr Time': 0, 'Pr Time': 0, 'Accuracy': 0, 'RAM': 0, 'VRAM': 0},
          'MobileNetV3': {'Tr Time': 0, 'Pr Time': 0, 'Accuracy': 0, 'RAM': 0, 'VRAM': 0}}
```

Modelo base

O modelo base será algum método que não seja um rede neural, para que possamos comparar ele com as técnicas mais robustas de MLP e CNN. Como base, usaremos algumas técnicas como Decision Tree, Random Forest e XGBoost.

> Note que nenhum desses modelos, originalmente, foram feitos para trabalhar com imagens. Para resolver esse problema, usaremos o método `ndarray.reshape`, para que as imagens (28 x 28) se tornem vetores (784). Os dados também serão escalonados para valores entre 0 e 1.

Decision Tree

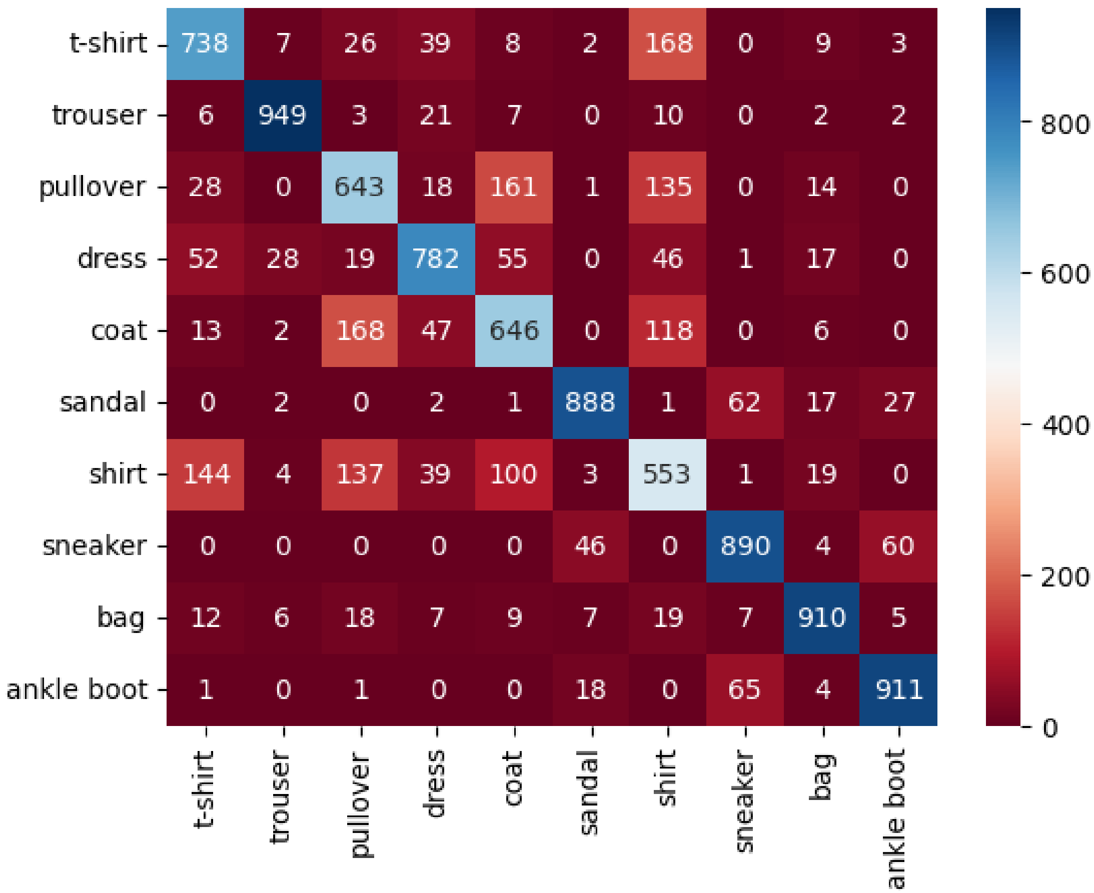
Árvores de decisão representam um modelo de tomada de decisão que é estruturado como uma árvore, na qual cada nó interno representa uma decisão com base em um atributo específico, cada ramo representa o resultado dessa decisão e cada folha representa a classe ou valor de saída.

```
class MyDecisionTree():
    def __init__(self, random_state=None):
        self.dtc = DecisionTreeClassifier(random_state=random_state)

    def flatten_scale(self, images):
        return images.reshape((images.shape[0], -1)) / 255

    def fit(self, images, labels):
        images_flat_scale = self.flatten_scale(images)
        self.dtc.fit(images_flat_scale, labels)

    def predict(self, images):
        images_flat_scale = self.flatten_scale(images)
        return self.dtc.predict(images_flat_scale)

decision_tree = MyDecisionTree(random_state=9)

time_0 = time.time()
decision_tree.fit(X_train, y_train)
time_f = time.time()
trtime_dt = (time_f-time_0)

time_0 = time.time()
y_test_pred_dt = decision_tree.predict(X_test)
time_f = time.time()
prtime_dt = (time_f-time_0)

print(classification_report(y_test, y_test_pred_dt))
print(f'Time spent: {trtime_dt:.2f}s')
models['Decision Tree']['Accuracy'] = accuracy_score(y_test,
y_test_pred_dt)
```

```python
models['Decision Tree']['Tr Time'] = trtime_dt
models['Decision Tree']['Pr Time'] = prtime_dt
```

```
              precision    recall  f1-score   support

           0       0.74      0.74      0.74      1000
           1       0.95      0.95      0.95      1000
           2       0.63      0.64      0.64      1000
           3       0.82      0.78      0.80      1000
           4       0.65      0.65      0.65      1000
           5       0.92      0.89      0.90      1000
           6       0.53      0.55      0.54      1000
           7       0.87      0.89      0.88      1000
           8       0.91      0.91      0.91      1000
           9       0.90      0.91      0.91      1000

    accuracy                           0.79     10000
   macro avg       0.79      0.79      0.79     10000
weighted avg       0.79      0.79      0.79     10000

Time spent: 39.36s
```

```python
cmatrix = confusion_matrix(y_test, y_test_pred_dt)
sns.heatmap(cmatrix, annot=True, fmt=".0f", cmap='RdBu',
xticklabels=labels_title, yticklabels=labels_title)
plt.show()
```



Random Forest

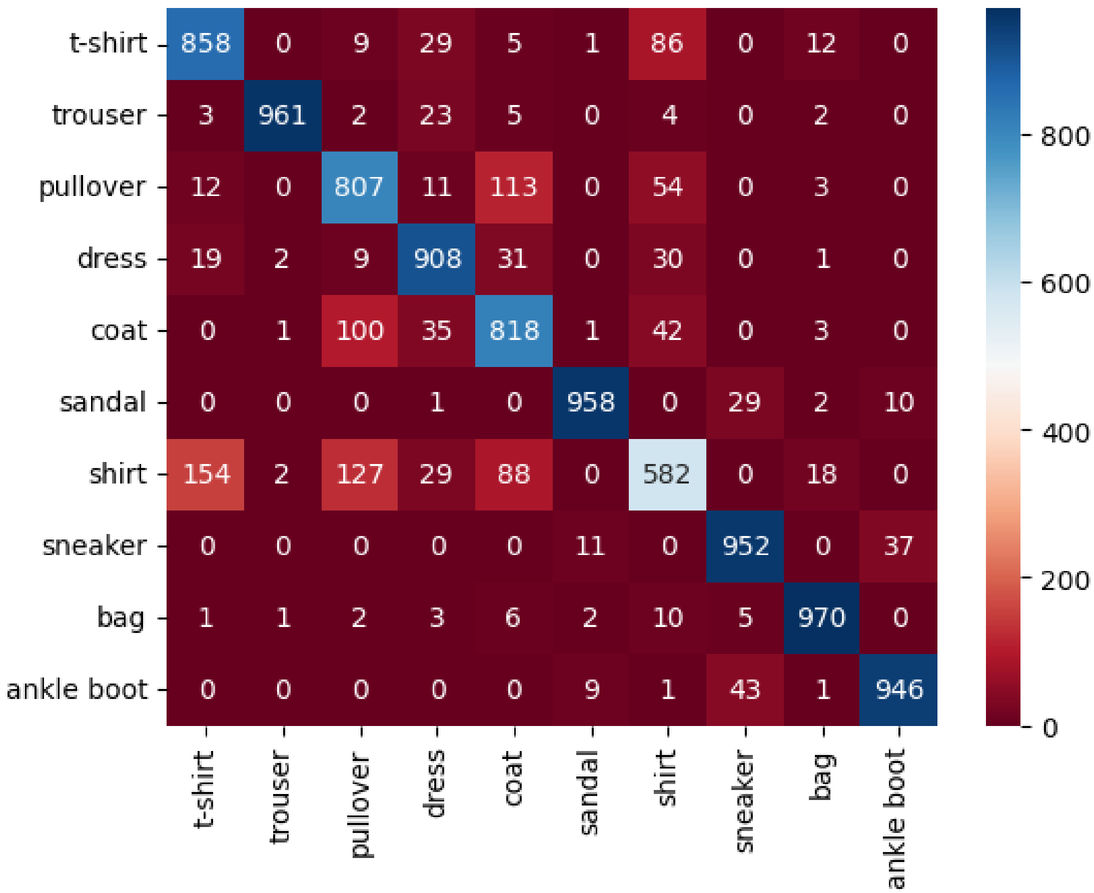
Random Forest é uma extensão das árvores de decisão que visa melhorar a robustez e a precisão do modelo por meio do conceito de ensemble learning, que combina vários modelos individuais para formar um modelo mais poderoso e geral. O processo de construção de uma Random Forest envolve a geração de várias árvores de decisão, cada uma treinada em uma amostra aleatória e independente dos dados originais.

```
class MyRandomForest():
    def __init__(self, random_state=None):
        self.rfc = RandomForestClassifier(random_state=random_state)

    def flatten_scale(self, images):
        return images.reshape((images.shape[0], -1)) / 255

    def fit(self, images, labels):
        images_flat_scale = self.flatten_scale(images)
        self.rfc.fit(images_flat_scale, labels)

    def predict(self, images):
        images_flat_scale = self.flatten_scale(images)
        return self.rfc.predict(images_flat_scale)
```

```python
random_forest = MyRandomForest(random_state=9)

time_0 = time.time()
random_forest.fit(X_train, y_train)
time_f = time.time()
trtime_rf = (time_f-time_0)

time_0 = time.time()
y_test_pred_rf = random_forest.predict(X_test)
time_f = time.time()
prtime_rf = (time_f-time_0)

print(classification_report(y_test, y_test_pred_rf))
print(f'Time spent: {trtime_rf:.2f}s')
models['Random Forest']['Accuracy'] = accuracy_score(y_test,
y_test_pred_rf)
models['Random Forest']['Tr Time'] = trtime_rf
models['Random Forest']['Pr Time'] = prtime_rf
```

```
              precision    recall  f1-score   support

           0       0.82      0.86      0.84      1000
           1       0.99      0.96      0.98      1000
           2       0.76      0.81      0.79      1000
           3       0.87      0.91      0.89      1000
           4       0.77      0.82      0.79      1000
           5       0.98      0.96      0.97      1000
           6       0.72      0.58      0.64      1000
           7       0.93      0.95      0.94      1000
           8       0.96      0.97      0.96      1000
           9       0.95      0.95      0.95      1000

    accuracy                           0.88     10000
   macro avg       0.88      0.88      0.87     10000
weighted avg       0.88      0.88      0.87     10000

Time spent: 94.37s
```

```python
cmatrix = confusion_matrix(y_test, y_test_pred_rf)
sns.heatmap(cmatrix, annot=True, fmt=".0f", cmap='RdBu',
xticklabels=labels_title, yticklabels=labels_title)
plt.show()
```



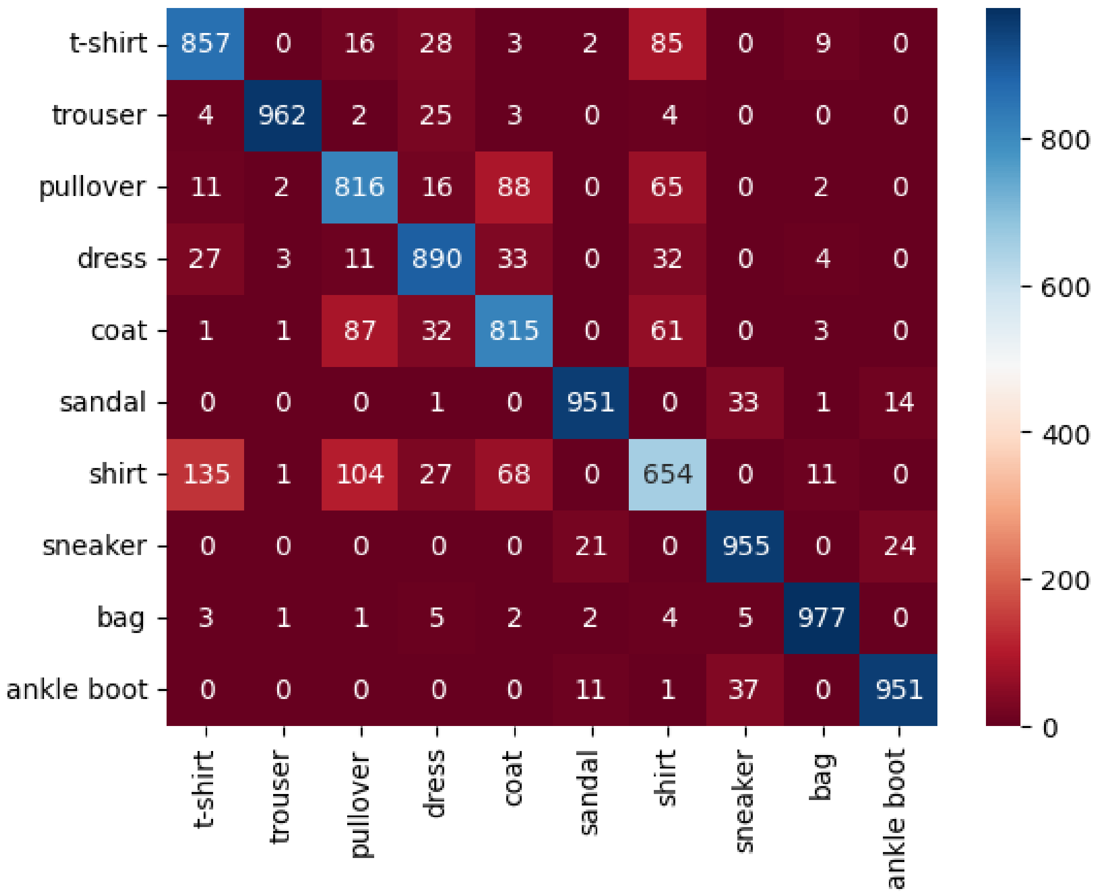
SVM

```
class MySVM():
    def __init__(self, random_state=None):
        self.svm = SVC(random_state=random_state)

    def flatten_scale(self, images):
        return images.reshape((images.shape[0], -1)) / 255

    def fit(self, images, labels):
        images_flat_scale = self.flatten_scale(images)
        self.svm.fit(images_flat_scale, labels)

    def predict(self, images):
        images_flat_scale = self.flatten_scale(images)
        return self.svm.predict(images_flat_scale)

svm = MySVM(random_state=9)

time_0 = time.time()
svm.fit(X_train, y_train)
time_f = time.time()
trtime_svm = (time_f-time_0)
```

```python
time_0 = time.time()
y_test_pred_svm = svm.predict(X_test)
time_f = time.time()
prtime_svm = (time_f-time_0)

print(classification_report(y_test, y_test_pred_svm))
print(f'Time spent: {trtime_svm:.2f}s')
models['SVM']['Accuracy'] = accuracy_score(y_test, y_test_pred_svm)
models['SVM']['Tr Time'] = trtime_svm
models['SVM']['Pr Time'] = prtime_svm
```

```
              precision    recall  f1-score   support

           0       0.83      0.86      0.84      1000
           1       0.99      0.96      0.98      1000
           2       0.79      0.82      0.80      1000
           3       0.87      0.89      0.88      1000
           4       0.81      0.81      0.81      1000
           5       0.96      0.95      0.96      1000
           6       0.72      0.65      0.69      1000
           7       0.93      0.95      0.94      1000
           8       0.97      0.98      0.97      1000
           9       0.96      0.95      0.96      1000

    accuracy                           0.88     10000
   macro avg       0.88      0.88      0.88     10000
weighted avg       0.88      0.88      0.88     10000

Time spent: 310.99s
```

```python
cmatrix = confusion_matrix(y_test, y_test_pred_svm)
sns.heatmap(cmatrix, annot=True, fmt=".0f", cmap='RdBu',
xticklabels=labels_title, yticklabels=labels_title)
plt.show()
```



MLP

A MLP (Multilayer Perceptron) pertence à categoria de modelos de aprendizado profundo. Consiste numa rede neural artificial onde os perceptrons são organizados em camadas. Assim como nos modelos base, MLPs também não lidam com entradas multidimensionais (como imagens), utilizaremos o método `nn.Flatten` do PyTorch para redimensionar a entrada para um vetor.

A rede será treinada por 15 épocas.

```
class MyMLP(nn.Module):
    def __init__(self, input_size, output_size):
        super(MyMLP, self).__init__()
        self.flatten = nn.Flatten()
        self.net = nn.Sequential(nn.Linear(input_size, 256),
nn.ReLU(),
                                 nn.Linear(256, 128), nn.ReLU(),
                                 nn.Linear(128, 64), nn.ReLU(),
nn.Dropout(p=0.5),
```

```
                                 nn.Linear(64, 32), nn.ReLU(),
nn.Dropout(p=0.25),
                                 nn.Linear(32, output_size))

    def forward(self, x):
        return self.net(self.flatten(x))

    def predict(self, x):
        return torch.argmax(self.forward(x), dim=1)

input_size = 28*28
output_size = 10

model_mlp = MyMLP(input_size, output_size).to(device)
loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model_mlp.parameters(), lr=0.001)

model_mlp
```

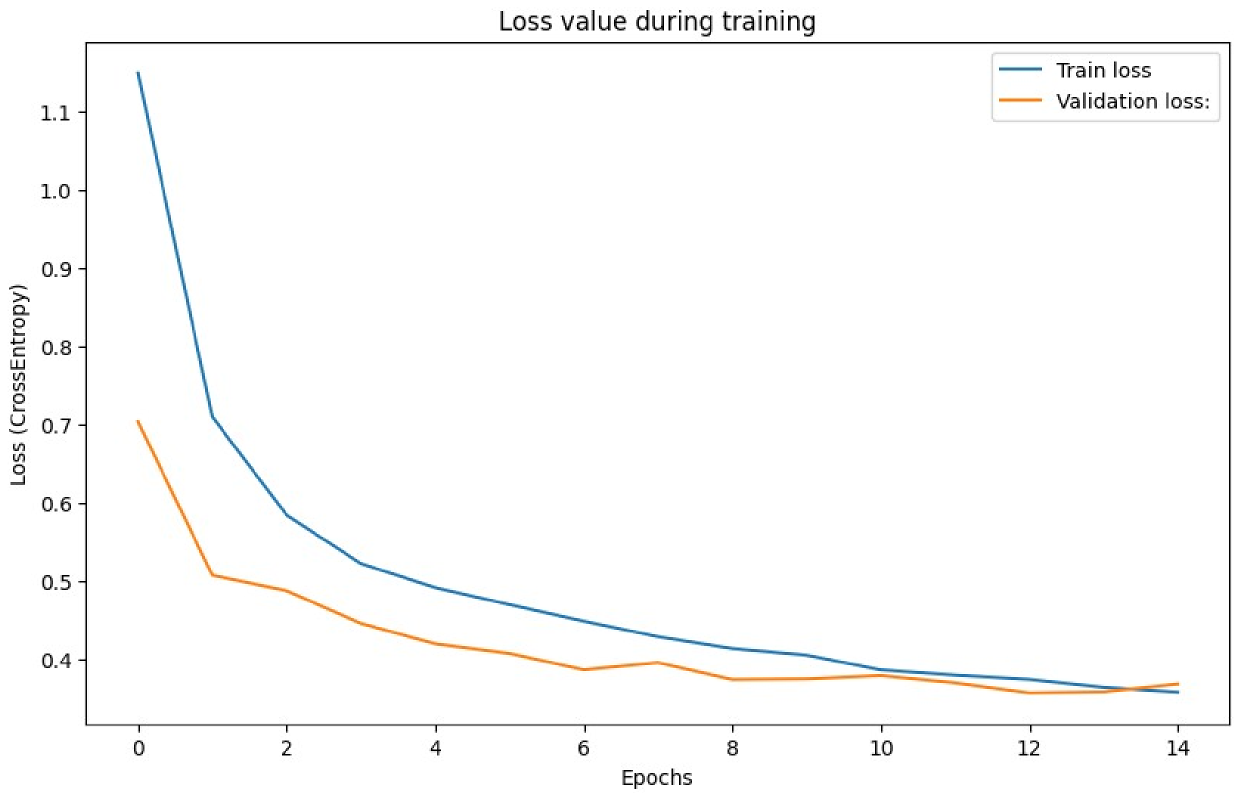
```
MyMLP(
  (flatten): Flatten(start_dim=1, end_dim=-1)
  (net): Sequential(
    (0): Linear(in_features=784, out_features=256, bias=True)
    (1): ReLU()
    (2): Linear(in_features=256, out_features=128, bias=True)
    (3): ReLU()
    (4): Linear(in_features=128, out_features=64, bias=True)
    (5): ReLU()
    (6): Dropout(p=0.5, inplace=False)
    (7): Linear(in_features=64, out_features=32, bias=True)
    (8): ReLU()
    (9): Dropout(p=0.25, inplace=False)
    (10): Linear(in_features=32, out_features=10, bias=True)
  )
)
```

```
trtime_mlp, ram_mlp, vram_mlp = train(model_mlp, train_mlp_cnn_loader,
val_mlp_cnn_loader, 15, loss_fn, optimizer)

y_test_pred_mlp, y_true, test_loss, prtime_mlp = test(model_mlp,
test_mlp_cnn_loader, loss_fn)
print(classification_report(y_true, y_test_pred_mlp))
print(f'Test loss: {test_loss:.4f}')
print(f'Time spent: {trtime_mlp:.2f}s')
models['MLP']['Accuracy'] = (accuracy_score(y_true, y_test_pred_mlp))
models['MLP']['Tr Time'] = trtime_mlp
models['MLP']['Pr Time'] = prtime_mlp
models['MLP']['RAM'] = ram_mlp
models['MLP']['VRAM'] = vram_mlp
```

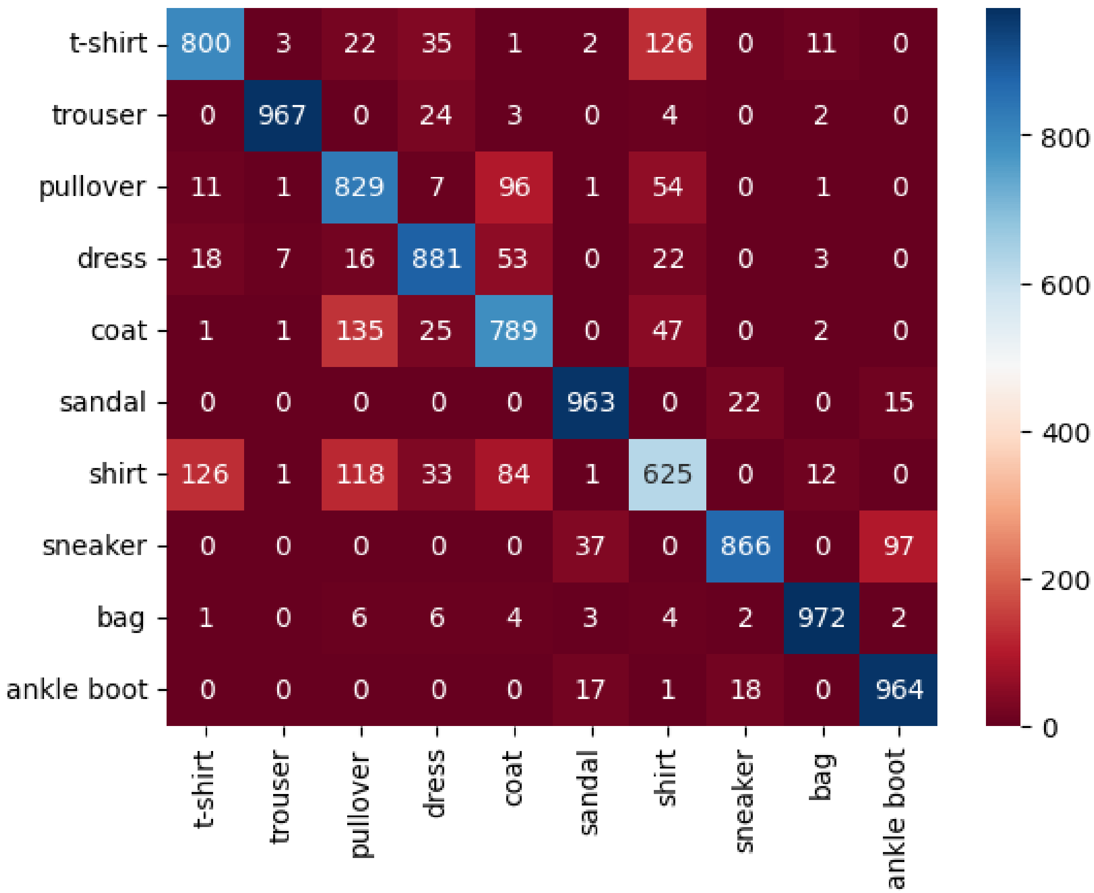
```
{"model_id":"999b494979e349b2925f69faad282630","version_major":2,"version_minor":0}

{"model_id":"695b08c2729742c59e7bf5e5241a9d00","version_major":2,"version_minor":0}

01: Train loss: 1.14873, RAM Usage: 2.4GB, VRAM Usage: 0.021GB |
Validation loss: 0.70286, Validation acc: 74.59%

{"model_id":"03c68114b39f44308b9893f2d0e66fa6","version_major":2,"version_minor":0}

{"model_id":"0cb0a709561f47af9164a2861ba3f2e2","version_major":2,"version_minor":0}

02: Train loss: 0.70898, RAM Usage: 2.4GB, VRAM Usage: 0.021GB |
Validation loss: 0.50661, Validation acc: 82.26%

{"model_id":"c1b4583303cc47158d7656b2e05e6a08","version_major":2,"version_minor":0}

{"model_id":"1ef197868c6f4544814545f83025f2fc","version_major":2,"version_minor":0}

03: Train loss: 0.58356, RAM Usage: 2.4GB, VRAM Usage: 0.021GB |
Validation loss: 0.48684, Validation acc: 82.84%

{"model_id":"37f24b3319ef4c0bafc15f1a54d20d08","version_major":2,"version_minor":0}

{"model_id":"519b98893b8e4c16ac72be6b32184a84","version_major":2,"version_minor":0}

04: Train loss: 0.52152, RAM Usage: 2.4GB, VRAM Usage: 0.021GB |
Validation loss: 0.44511, Validation acc: 84.32%

{"model_id":"c2183abd9db34eb58c893869b9bf8f56","version_major":2,"version_minor":0}

{"model_id":"43befda3c40f44f0b39a3d0a1509ad63","version_major":2,"version_minor":0}

05: Train loss: 0.49080, RAM Usage: 2.4GB, VRAM Usage: 0.021GB |
Validation loss: 0.41903, Validation acc: 85.27%

{"model_id":"54bc8aa1bee44fda924333754cead2d6","version_major":2,"version_minor":0}

{"model_id":"bfc4e0abe9b747d982eab591ecf0f88a","version_major":2,"version_minor":0}

06: Train loss: 0.46917, RAM Usage: 2.3GB, VRAM Usage: 0.021GB |
Validation loss: 0.40675, Validation acc: 86.08%
```

```
{"model_id":"fddca3f45a86403cb58077fa5a307be3","version_major":2,"version_minor":0}

{"model_id":"a0189f6111cf4d96adba787a3c99c3aa","version_major":2,"version_minor":0}

07: Train loss: 0.44795, RAM Usage: 2.4GB, VRAM Usage: 0.021GB | Validation loss: 0.38587, Validation acc: 86.46%

{"model_id":"ed30fa000d69457f99086ed82abb1217","version_major":2,"version_minor":0}

{"model_id":"0292c9064b3049f5bc0cf318680a38f5","version_major":2,"version_minor":0}

08: Train loss: 0.42813, RAM Usage: 2.4GB, VRAM Usage: 0.021GB | Validation loss: 0.39498, Validation acc: 85.84%

{"model_id":"10ab7dd5755f486daa072f37408a0c4a","version_major":2,"version_minor":0}

{"model_id":"19878e5174fb40a8918adfb69db1ceb4","version_major":2,"version_minor":0}

09: Train loss: 0.41300, RAM Usage: 2.4GB, VRAM Usage: 0.021GB | Validation loss: 0.37344, Validation acc: 86.63%

{"model_id":"778cb51580e0477d96efc4284448d9bd","version_major":2,"version_minor":0}

{"model_id":"ac33ca55861e4f2f8f37cbb2294726c9","version_major":2,"version_minor":0}

10: Train loss: 0.40456, RAM Usage: 2.4GB, VRAM Usage: 0.021GB | Validation loss: 0.37420, Validation acc: 87.29%

{"model_id":"921bb5d628d54d22a16ac485f303f6ad","version_major":2,"version_minor":0}

{"model_id":"fc309df2ea064235a881ee553881e124","version_major":2,"version_minor":0}

11: Train loss: 0.38572, RAM Usage: 2.4GB, VRAM Usage: 0.021GB | Validation loss: 0.37845, Validation acc: 86.69%

{"model_id":"0586f7e8126843838687df41bab10742","version_major":2,"version_minor":0}

{"model_id":"84cbf9991f034f849eafe3b256546ee0","version_major":2,"version_minor":0}

12: Train loss: 0.37907, RAM Usage: 2.4GB, VRAM Usage: 0.021GB | Validation loss: 0.36921, Validation acc: 87.04%
```

```
{"model_id":"bb0f67d4ddc34459ba505b484a61a11a","version_major":2,"version_minor":0}

{"model_id":"e6f9df2f39fd4974a404af17a7d02f75","version_major":2,"version_minor":0}

13: Train loss: 0.37354, RAM Usage: 2.4GB, VRAM Usage: 0.021GB |
Validation loss: 0.35637, Validation acc: 87.22%

{"model_id":"5b0da71128c14108b0a8093e19c7dbef","version_major":2,"version_minor":0}

{"model_id":"2ed6f277db8f4be2ac1715d325dccaa1","version_major":2,"version_minor":0}

14: Train loss: 0.36358, RAM Usage: 2.4GB, VRAM Usage: 0.021GB |
Validation loss: 0.35773, Validation acc: 87.76%

{"model_id":"1c4bc34695df4a1392a18b79bbd7035a","version_major":2,"version_minor":0}

{"model_id":"2646250217624e24b0960aa0781deb7d","version_major":2,"version_minor":0}

15: Train loss: 0.35728, RAM Usage: 2.4GB, VRAM Usage: 0.021GB |
Validation loss: 0.36777, Validation acc: 87.31%

{"model_id":"22d8e3829c5543da9f97934ee41065da","version_major":2,"version_minor":0}

              precision    recall  f1-score   support

           0       0.84      0.80      0.82      1000
           1       0.99      0.97      0.98      1000
           2       0.74      0.83      0.78      1000
           3       0.87      0.88      0.88      1000
           4       0.77      0.79      0.78      1000
           5       0.94      0.96      0.95      1000
           6       0.71      0.62      0.66      1000
           7       0.95      0.87      0.91      1000
           8       0.97      0.97      0.97      1000
           9       0.89      0.96      0.93      1000

    accuracy                           0.87     10000
   macro avg       0.87      0.87      0.86     10000
weighted avg       0.87      0.87      0.86     10000

Test loss: 0.3870
Time spent: 590.05s
```




```
cmatrix = confusion_matrix(y_true, y_test_pred_mlp)
sns.heatmap(cmatrix, annot=True, fmt=".0f", cmap='RdBu',
xticklabels=labels_title, yticklabels=labels_title)
plt.show()
```



Rede Convolucional

Finalmente usaremos a primeira arquitetura proposta para, de fato, trabalhar com dados estruturados em forma de grade, especialmente em tarefas relacionadas a imagens. As CNNs conseguem estrair relações de localidade nos dados e ainda usam menos parâmetros que uma ANN do mesmo porte, alem de podermos utilizar os recursos de GPU disponíveis.

A rede convolucional será treinada por 15 épocas.

```
class MyCNN(nn.Module):
    def __init__(self, in_channels=1, out_classes=10):
        super(MyCNN, self).__init__()
        self.convs = nn.Sequential(nn.Conv2d(in_channels=in_channels,
# 1x28x28
                                             out_channels=4,
                                             kernel_size=3,
                                             padding=1),
                                   nn.BatchNorm2d(num_features=4),
                                   nn.ReLU(inplace=True),  # 4x28x28
```

```
                                   nn.MaxPool2d(kernel_size=2, 
stride=2),

                                   nn.Conv2d(in_channels=4,  # 4x14x14
                                             out_channels=8,
                                             kernel_size=3,
                                             padding=1),
                                   nn.BatchNorm2d(num_features=8),
                                   nn.ReLU(inplace=True),  # 8x14x14
                                   nn.MaxPool2d(kernel_size=2, 
stride=2))  # 8x7x7
        self.flatten = nn.Flatten()
        self.net = nn.Sequential(nn.Linear(8*7*7, 128), nn.ReLU(),
                                 nn.Linear(128, 64), nn.ReLU(), 
nn.Dropout(p=0.5),
                                 nn.Linear(64, 32), nn.ReLU(), 
nn.Dropout(p=0.25),
                                 nn.Linear(32, output_size))

    def forward(self, x):
        return self.net(self.flatten(self.convs(x)))

    def predict(self, x):
        return torch.argmax(self.forward(x), dim=1)

output_size = 10

model_cnn = MyCNN(1, output_size).to(device)
loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model_cnn.parameters(), lr=0.001)

model_cnn
```

```
MyCNN(
  (convs): Sequential(
    (0): Conv2d(1, 4, kernel_size=(3, 3), stride=(1, 1), padding=(1, 
1))
    (1): BatchNorm2d(4, eps=1e-05, momentum=0.1, affine=True, 
track_running_stats=True)
    (2): ReLU(inplace=True)
    (3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, 
ceil_mode=False)
    (4): Conv2d(4, 8, kernel_size=(3, 3), stride=(1, 1), padding=(1, 
1))
    (5): BatchNorm2d(8, eps=1e-05, momentum=0.1, affine=True, 
track_running_stats=True)
    (6): ReLU(inplace=True)
    (7): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, 
ceil_mode=False)
  )
```

```
  (flatten): Flatten(start_dim=1, end_dim=-1)
  (net): Sequential(
    (0): Linear(in_features=392, out_features=128, bias=True)
    (1): ReLU()
    (2): Linear(in_features=128, out_features=64, bias=True)
    (3): ReLU()
    (4): Dropout(p=0.5, inplace=False)
    (5): Linear(in_features=64, out_features=32, bias=True)
    (6): ReLU()
    (7): Dropout(p=0.25, inplace=False)
    (8): Linear(in_features=32, out_features=10, bias=True)
  )
)
```

```
trtime_cnn, ram_cnn, vram_cnn = train(model_cnn, train_mlp_cnn_loader,
val_mlp_cnn_loader, 15, loss_fn, optimizer)

y_test_pred_cnn, y_true, test_loss, prtime_cnn = test(model_cnn,
test_mlp_cnn_loader, loss_fn)
print(classification_report(y_true, y_test_pred_cnn))
print(f'Test loss: {test_loss:.4f}')
print(f'Time spent: {trtime_cnn:.2f}s')
models['Custom CNN']['Accuracy'] = (accuracy_score(y_true,
y_test_pred_cnn))
models['Custom CNN']['Tr Time'] = trtime_cnn
models['Custom CNN']['Pr Time'] = prtime_cnn
models['Custom CNN']['RAM'] = ram_cnn
models['Custom CNN']['VRAM'] = vram_cnn
```

{"model_id":"e739904bd22a452e92b0d53e3f609520","version_major":2,"version_minor":0}

{"model_id":"f9b451874d504fd1822965799ef0e39a","version_major":2,"version_minor":0}

```
01: Train loss: 0.98486, RAM Usage: 2.5GB, VRAM Usage: 0.028GB |
Validation loss: 0.52172, Validation acc: 80.50%
```

{"model_id":"483db8234d77445cb8a4bca6aa272a45","version_major":2,"version_minor":0}

{"model_id":"e0e194f59adb427c95daf674eca62d00","version_major":2,"version_minor":0}

```
02: Train loss: 0.57361, RAM Usage: 2.5GB, VRAM Usage: 0.028GB |
Validation loss: 0.43979, Validation acc: 84.39%
```

{"model_id":"a241d88a8c7440f3984576c06ae27a56","version_major":2,"version_minor":0}

```
{"model_id":"64a4344b5b5c4417989329d50ec283e5","version_major":2,"version_minor":0}

03: Train loss: 0.49295, RAM Usage: 2.5GB, VRAM Usage: 0.028GB |
Validation loss: 0.38360, Validation acc: 85.88%

{"model_id":"d0de9709fd2f47deb275633fc18b5136","version_major":2,"version_minor":0}

{"model_id":"4ee9e3fa158f4f8195f9e0586f059dd2","version_major":2,"version_minor":0}

04: Train loss: 0.45786, RAM Usage: 2.5GB, VRAM Usage: 0.028GB |
Validation loss: 0.37053, Validation acc: 86.44%

{"model_id":"3b123a0567e54955b79abea8b21490bb","version_major":2,"version_minor":0}

{"model_id":"5ee473f58cf24e10823d72b5839327c0","version_major":2,"version_minor":0}

05: Train loss: 0.42525, RAM Usage: 2.5GB, VRAM Usage: 0.028GB |
Validation loss: 0.35827, Validation acc: 87.16%

{"model_id":"2ca5a8fedc8941bc81978b70866b7348","version_major":2,"version_minor":0}

{"model_id":"d18921ed45464ad1ab0ceb3d012af7ba","version_major":2,"version_minor":0}

06: Train loss: 0.40760, RAM Usage: 2.5GB, VRAM Usage: 0.028GB |
Validation loss: 0.34539, Validation acc: 87.43%

{"model_id":"6eab6111f6194305848de3f12d444a16","version_major":2,"version_minor":0}

{"model_id":"b06f2bfa441d4a03aaf6c339dfad6496","version_major":2,"version_minor":0}

07: Train loss: 0.38887, RAM Usage: 2.5GB, VRAM Usage: 0.028GB |
Validation loss: 0.32637, Validation acc: 88.19%

{"model_id":"56a70aacc49640f187ed78a61321ae4d","version_major":2,"version_minor":0}

{"model_id":"791c72277acd402d8529fbffa0ee531f","version_major":2,"version_minor":0}

08: Train loss: 0.37108, RAM Usage: 2.5GB, VRAM Usage: 0.028GB |
Validation loss: 0.32565, Validation acc: 88.08%

{"model_id":"c2e1182d6cea4df39421e5b62d71faaf","version_major":2,"version_minor":0}
```

{"model_id":"d0bf5a8614b24cd8a3267cd81b19b95b","version_major":2,"version_minor":0}

```
09: Train loss: 0.36487, RAM Usage: 2.5GB, VRAM Usage: 0.028GB |
Validation loss: 0.33433, Validation acc: 87.75%
```

{"model_id":"d58e6201f6724b2fab280ed33c5cf16c","version_major":2,"version_minor":0}

{"model_id":"09f60ebf7354417e87aa879ab935cda3","version_major":2,"version_minor":0}

```
10: Train loss: 0.35530, RAM Usage: 2.5GB, VRAM Usage: 0.028GB |
Validation loss: 0.33569, Validation acc: 87.68%
```

{"model_id":"bcb8f25df3204e71a8d0204e386d007e","version_major":2,"version_minor":0}

{"model_id":"5dc702ab28504a5d8733d9ed3b12c35d","version_major":2,"version_minor":0}

```
11: Train loss: 0.34528, RAM Usage: 2.5GB, VRAM Usage: 0.028GB |
Validation loss: 0.32688, Validation acc: 88.44%
```

{"model_id":"c0e07f6ea61e4ea58c34aacf51fc0844","version_major":2,"version_minor":0}

{"model_id":"a3fd6196666a4914a0a695248ee751fd","version_major":2,"version_minor":0}

```
12: Train loss: 0.33226, RAM Usage: 2.5GB, VRAM Usage: 0.028GB |
Validation loss: 0.33455, Validation acc: 88.07%
```

{"model_id":"ebcc380aee7744eb944625ff03db872f","version_major":2,"version_minor":0}

{"model_id":"8483d640d8f04afe8eea0382a5541065","version_major":2,"version_minor":0}

```
13: Train loss: 0.32504, RAM Usage: 2.5GB, VRAM Usage: 0.028GB |
Validation loss: 0.31472, Validation acc: 88.93%
```

{"model_id":"7f32668ced2d456bb8f2a2eba7d24e5b","version_major":2,"version_minor":0}

{"model_id":"38ca7f6accd742258ebecb9eac25aa90","version_major":2,"version_minor":0}

```
14: Train loss: 0.31755, RAM Usage: 2.5GB, VRAM Usage: 0.028GB |
Validation loss: 0.30506, Validation acc: 89.37%
```

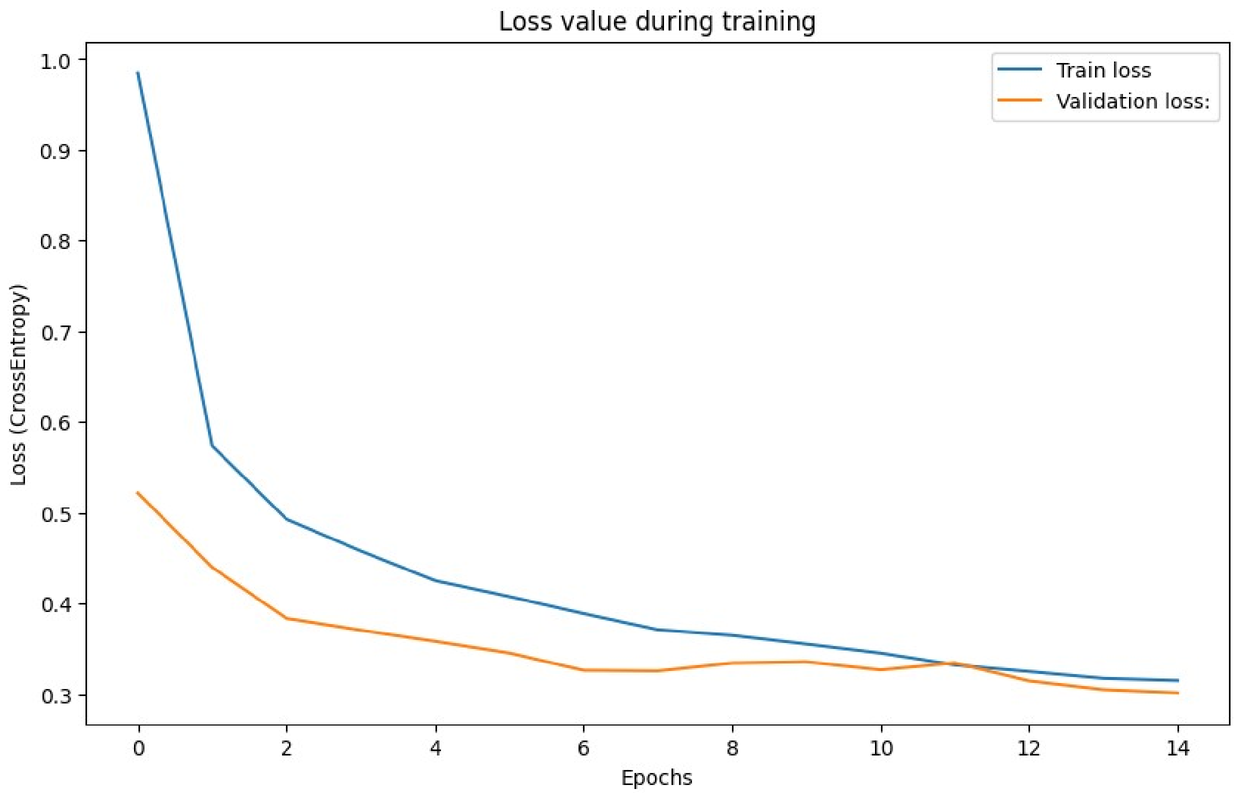
{"model_id":"3997186458694ba2ae257d15d50b2fa8","version_major":2,"version_minor":0}

{"model_id":"00917fb9ab0c435aa03cab7f04303770","version_major":2,"version_minor":0}

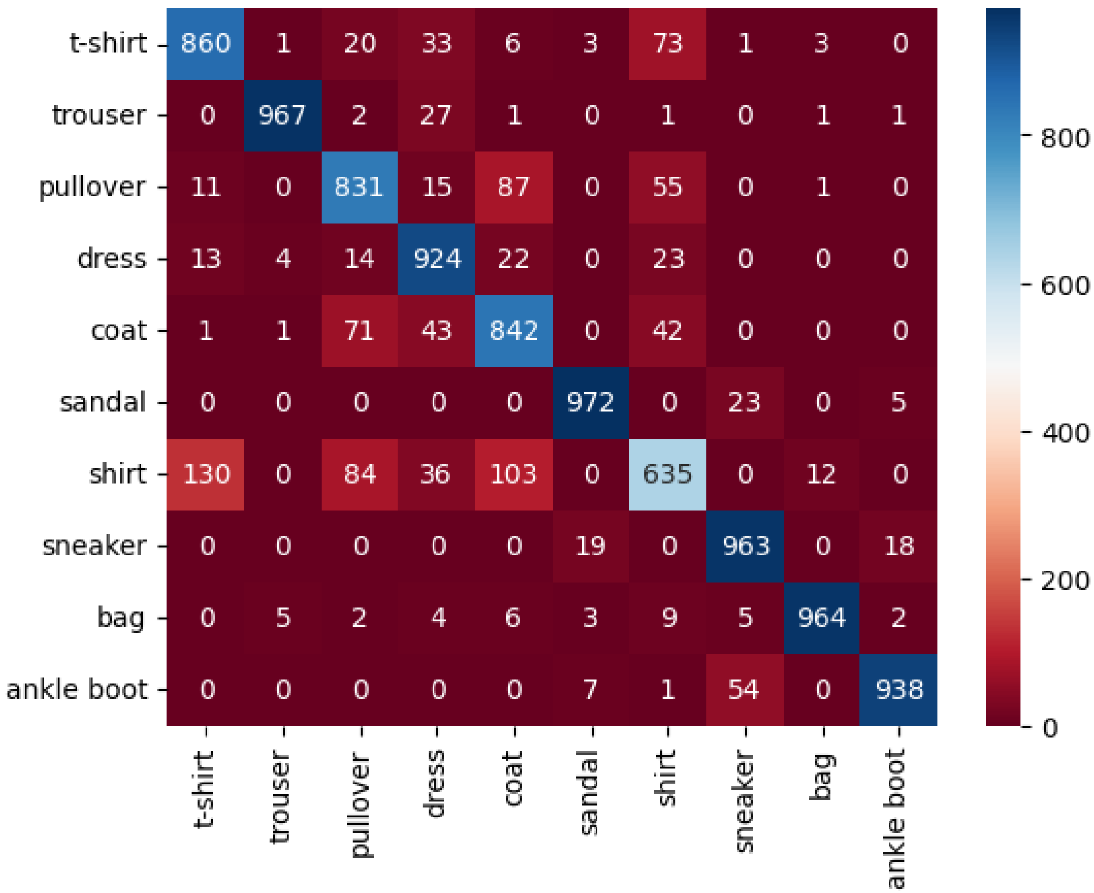
```
15: Train loss: 0.31511, RAM Usage: 2.5GB, VRAM Usage: 0.028GB |
Validation loss: 0.30149, Validation acc: 89.34%
```

{"model_id":"4d65700cc97d4114ac09b0cd03a7474b","version_major":2,"version_minor":0}

```
              precision    recall  f1-score   support

           0       0.85      0.86      0.85      1000
           1       0.99      0.97      0.98      1000
           2       0.81      0.83      0.82      1000
           3       0.85      0.92      0.89      1000
           4       0.79      0.84      0.81      1000
           5       0.97      0.97      0.97      1000
           6       0.76      0.64      0.69      1000
           7       0.92      0.96      0.94      1000
           8       0.98      0.96      0.97      1000
           9       0.97      0.94      0.96      1000

    accuracy                           0.89     10000
   macro avg       0.89      0.89      0.89     10000
weighted avg       0.89      0.89      0.89     10000

Test loss: 0.3217
Time spent: 625.57s
```




```
cmatrix = confusion_matrix(y_true, y_test_pred_cnn)
sns.heatmap(cmatrix, annot=True, fmt=".0f", cmap='RdBu',
xticklabels=labels_title, yticklabels=labels_title)
plt.show()
```



Rede Convolucional consolidada

Para essa seção, usaremos as redes VGG-16, ResNet-34 e MobileNetV3 famosas por seus desempenhos na competição ImageNet e usadas até hoje como opções de transfer learning. Para tal, precisaremos mudar a primeira e ultima camada de cada modelo, já que temos 10 classes e 1 canal de entrada (o que não ocorre nos modelos originais).

> Todas as redes consolidadas serão treinadas por 5 épocas.

VGG-16

A VGG-16 é uma arquitetura de rede neural convolucional desenvolvida pela Visual Geometry Group (VGG). A VGG-16 é conhecida por sua simplicidade relativa, apresentando convoluções 3x3 consecutivas em várias camadas, seguidas por camadas de pooling.

> Cada agrupamento de convoluções seguidas por pooling configura um bloco VGG. A quantidade de blocos define qual rede da família será usada.

```
class MyVGG16(nn.Module):
    def __init__(self, in_channels=1, out_classes=10):
        super(MyVGG16, self).__init__()
        self.vgg = vgg16(weights=VGG16_Weights.DEFAULT)
        self.vgg.features[0] = nn.Conv2d(1, 64, kernel_size=(3, 3),
stride=(1, 1), padding=(1, 1))
        self.vgg.classifier[-1] = nn.Linear(in_features=4096,
out_features=10, bias=True)

    def forward(self, x):
        return self.vgg(x)

    def predict(self, x):
        return torch.argmax(self.forward(x), dim=1)

output_size = 10

model_vgg = MyVGG16(1, output_size).to(device)
loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model_vgg.parameters(), lr=0.0001)

model_vgg
```

```
Downloading: "https://download.pytorch.org/models/vgg16-397923af.pth"
to /root/.cache/torch/hub/checkpoints/vgg16-397923af.pth
100%|██████████| 528M/528M [00:07<00:00, 70.1MB/s]
```

```
MyVGG16(
  (vgg): VGG(
    (features): Sequential(
      (0): Conv2d(1, 64, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
      (1): ReLU(inplace=True)
      (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
      (3): ReLU(inplace=True)
      (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
      (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
      (6): ReLU(inplace=True)
      (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
      (8): ReLU(inplace=True)
      (9): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
      (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
      (11): ReLU(inplace=True)
      (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
```

```
padding=(1, 1))
      (13): ReLU(inplace=True)
      (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
      (15): ReLU(inplace=True)
      (16): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
      (17): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
      (18): ReLU(inplace=True)
      (19): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
      (20): ReLU(inplace=True)
      (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
      (22): ReLU(inplace=True)
      (23): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
      (24): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
      (25): ReLU(inplace=True)
      (26): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
      (27): ReLU(inplace=True)
      (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
      (29): ReLU(inplace=True)
      (30): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
    )
    (avgpool): AdaptiveAvgPool2d(output_size=(7, 7))
    (classifier): Sequential(
      (0): Linear(in_features=25088, out_features=4096, bias=True)
      (1): ReLU(inplace=True)
      (2): Dropout(p=0.5, inplace=False)
      (3): Linear(in_features=4096, out_features=4096, bias=True)
      (4): ReLU(inplace=True)
      (5): Dropout(p=0.5, inplace=False)
      (6): Linear(in_features=4096, out_features=10, bias=True)
    )
  )
)

trtime_vgg, ram_vgg, vram_vgg = train(model_vgg,
train_cnn_cons_loader, val_cnn_cons_loader, 5, loss_fn, optimizer)

y_test_pred_vgg, y_true, test_loss, prtime_vgg = test(model_vgg,
test_cnn_cons_loader, loss_fn)
print(classification_report(y_true, y_test_pred_vgg))
print(f'Test loss: {test_loss:.4f}')
```

```python
print(f'Time spent: {trtime_vgg:.2f}s')
models['VGG16']['Accuracy'] = (accuracy_score(y_true,
y_test_pred_vgg))
models['VGG16']['Tr Time'] = trtime_vgg
models['VGG16']['Pr Time'] = prtime_vgg
models['VGG16']['RAM'] = ram_vgg
models['VGG16']['VRAM'] = vram_vgg
```

{"model_id":"b34a3da44c394c5788247e14e80a6c04","version_major":2,"version_minor":0}

{"model_id":"92545942ff4142a7bd68e8d647a9b239","version_major":2,"version_minor":0}

```
01: Train loss: 0.43226, RAM Usage: 2.6GB, VRAM Usage: 7.8GB |
Validation loss: 0.24820, Validation acc: 90.92%
```

{"model_id":"05def1e56414462d930825a7d089cf4b","version_major":2,"version_minor":0}

{"model_id":"35b0b980f2ad429f984b42427194dece","version_major":2,"version_minor":0}

```
02: Train loss: 0.24344, RAM Usage: 2.6GB, VRAM Usage: 7.8GB |
Validation loss: 0.21469, Validation acc: 92.10%
```

{"model_id":"0e861525497d44bf91aa019485518af8","version_major":2,"version_minor":0}

{"model_id":"3f3d7e7bdb8444da880283ba7488ee72","version_major":2,"version_minor":0}

```
03: Train loss: 0.20038, RAM Usage: 2.6GB, VRAM Usage: 7.8GB |
Validation loss: 0.20606, Validation acc: 92.46%
```

{"model_id":"ec84e944799c40ec87f6df05f92c2384","version_major":2,"version_minor":0}

{"model_id":"bbd020f433464eafb82b7b31ea225717","version_major":2,"version_minor":0}

```
04: Train loss: 0.17367, RAM Usage: 2.6GB, VRAM Usage: 7.8GB |
Validation loss: 0.19358, Validation acc: 92.95%
```

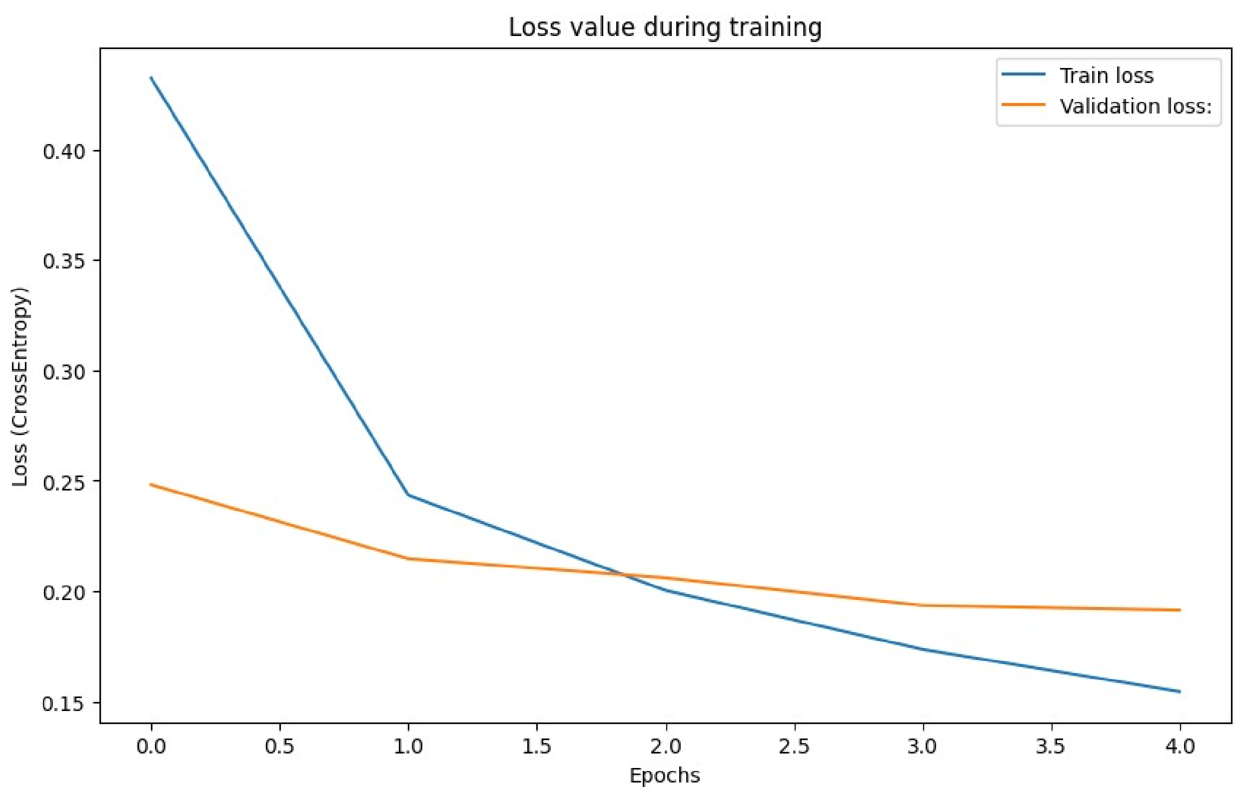
{"model_id":"eb6aa884b68e49dc81378ec33454afad","version_major":2,"version_minor":0}

{"model_id":"a04995c8f1174df1943011c47e69bfef","version_major":2,"version_minor":0}

```
05: Train loss: 0.15461, RAM Usage: 2.6GB, VRAM Usage: 7.8GB |
Validation loss: 0.19140, Validation acc: 93.13%
```

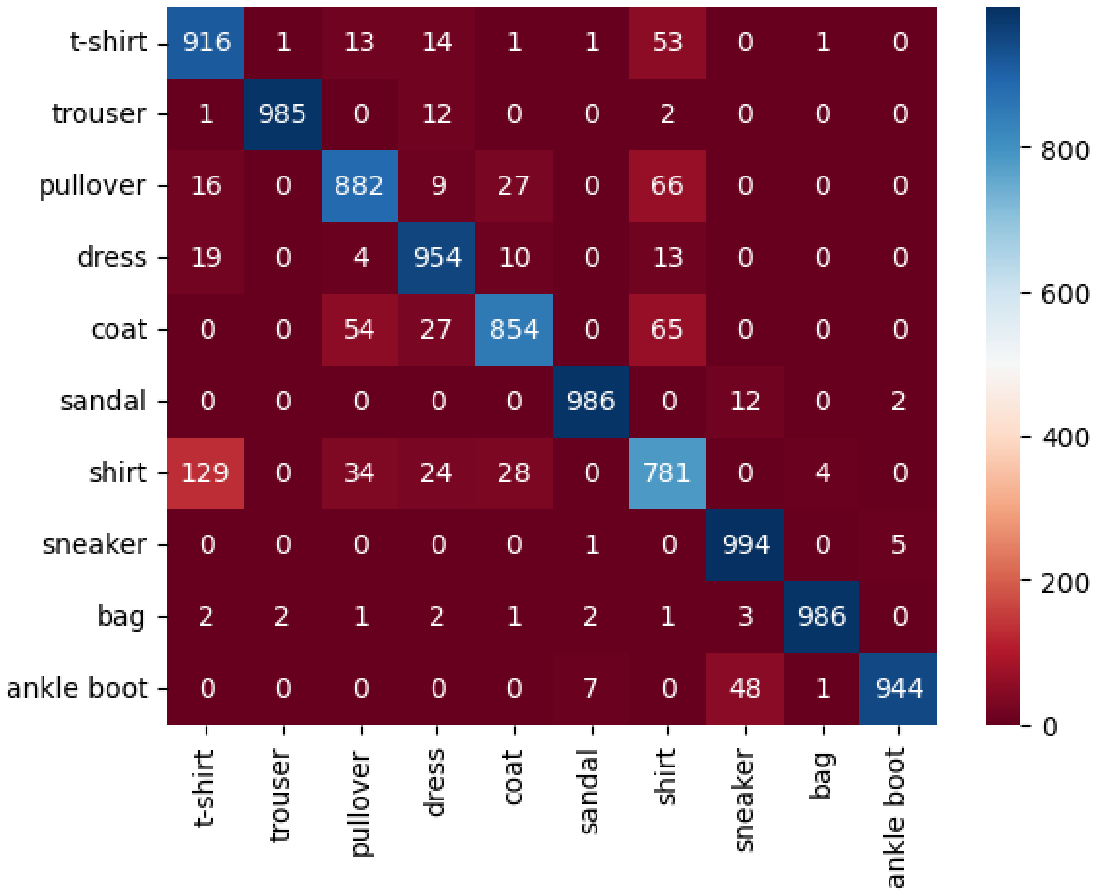
{"model_id":"737dc273c8544386928e6b9d4dea0a99","version_major":2,"version_minor":0}

```
              precision    recall  f1-score   support

           0       0.85      0.92      0.88      1000
           1       1.00      0.98      0.99      1000
           2       0.89      0.88      0.89      1000
           3       0.92      0.95      0.93      1000
           4       0.93      0.85      0.89      1000
           5       0.99      0.99      0.99      1000
           6       0.80      0.78      0.79      1000
           7       0.94      0.99      0.97      1000
           8       0.99      0.99      0.99      1000
           9       0.99      0.94      0.97      1000

    accuracy                           0.93     10000
   macro avg       0.93      0.93      0.93     10000
weighted avg       0.93      0.93      0.93     10000

Test loss: 0.2019
Time spent: 3668.58s
```




```
cmatrix = confusion_matrix(y_true, y_test_pred_vgg)
sns.heatmap(cmatrix, annot=True, fmt=".0f", cmap='RdBu',
```

```
xticklabels=labels_title, yticklabels=labels_title)
plt.show()
```



ResNet-34

A ResNet-34 é uma variação da arquitetura ResNet (Residual Network). A principal inovação das redes ResNet é a introdução de blocos residuais, que ajudam a superar o problema de degradação do desempenho observado em redes mais profundas.

> O conceito central da ResNet é o uso de blocos residuais, que introduzem conexões de atalho (skip connections) para pular uma ou mais camadas. Essas conexões permitem que o gradiente flua diretamente através do bloco, facilitando o treinamento de redes muito profundas.

```
class MyResNet34(nn.Module):
    def __init__(self, in_channels=1, out_classes=10):
        super(MyResNet34, self).__init__()
        self.rn34 = resnet34(weights=ResNet34_Weights.DEFAULT)
        self.rn34.conv1 = nn.Conv2d(1, 64, kernel_size=(7, 7),
stride=(2, 2), padding=(3, 3), bias=False)
        self.rn34.fc = nn.Linear(in_features=512, out_features=10,
bias=True)
```

```
    def forward(self, x):
        return self.rn34(x)

    def predict(self, x):
        return torch.argmax(self.forward(x), dim=1)

output_size = 10

model_rn34 = MyResNet34(1, output_size).to(device)
loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model_rn34.parameters(), lr=0.0001)

model_rn34
```

```
Downloading: "https://download.pytorch.org/models/resnet34-
b627a593.pth" to /root/.cache/torch/hub/checkpoints/resnet34-
b627a593.pth
100%|██████████| 83.3M/83.3M [00:00<00:00, 143MB/s]

MyResNet34(
  (rn34): ResNet(
    (conv1): Conv2d(1, 64, kernel_size=(7, 7), stride=(2, 2),
padding=(3, 3), bias=False)
    (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (relu): ReLU(inplace=True)
    (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1,
dilation=1, ceil_mode=False)
    (layer1): Sequential(
      (0): BasicBlock(
        (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
        (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
      (1): BasicBlock(
        (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
        (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
```

```
track_running_stats=True)
      )
      (2): BasicBlock(
        (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
        (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (layer2): Sequential(
      (0): BasicBlock(
        (conv1): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2),
padding=(1, 1), bias=False)
        (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (downsample): Sequential(
          (0): Conv2d(64, 128, kernel_size=(1, 1), stride=(2, 2),
bias=False)
          (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        )
      )
      (1): BasicBlock(
        (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
        (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
      (2): BasicBlock(
        (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
        (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (relu): ReLU(inplace=True)
```

```
        (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
      (3): BasicBlock(
        (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
        (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (layer3): Sequential(
      (0): BasicBlock(
        (conv1): Conv2d(128, 256, kernel_size=(3, 3), stride=(2, 2),
padding=(1, 1), bias=False)
        (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (downsample): Sequential(
          (0): Conv2d(128, 256, kernel_size=(1, 1), stride=(2, 2),
bias=False)
          (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        )
      )
      (1): BasicBlock(
        (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
        (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
      (2): BasicBlock(
        (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
```

```
        (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
      (3): BasicBlock(
        (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
        (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
      (4): BasicBlock(
        (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
        (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
      (5): BasicBlock(
        (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
        (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (layer4): Sequential(
      (0): BasicBlock(
        (conv1): Conv2d(256, 512, kernel_size=(3, 3), stride=(2, 2),
padding=(1, 1), bias=False)
        (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (relu): ReLU(inplace=True)
```

```
        (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (downsample): Sequential(
          (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2),
bias=False)
          (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        )
      )
      (1): BasicBlock(
        (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
        (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
      (2): BasicBlock(
        (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
        (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
    (fc): Linear(in_features=512, out_features=10, bias=True)
  )
)

trtime_rn34, ram_rn34, vram_rn34 = train(model_rn34,
train_cnn_cons_loader, val_cnn_cons_loader, 5, loss_fn, optimizer)

y_test_pred_rn34, y_true, test_loss, prtime_rn34 = test(model_rn34,
test_cnn_cons_loader, loss_fn)
print(classification_report(y_true, y_test_pred_rn34))
print(f'Test loss: {test_loss:.4f}')
print(f'Time spent: {trtime_rn34:.2f}s')
models['ResNet34']['Accuracy'] = (accuracy_score(y_true,
y_test_pred_rn34))
models['ResNet34']['Tr Time'] = trtime_rn34
```

```python
models['ResNet34']['Pr Time'] = prtime_rn34
models['ResNet34']['RAM'] = ram_rn34
models['ResNet34']['VRAM'] = vram_rn34
```

{"model_id":"4e69a1f1272041e5ad28d55bd6c30d79","version_major":2,"version_minor":0}

{"model_id":"986404a5317f4246b20cb2feccb21d5c","version_major":2,"version_minor":0}

```
01: Train loss: 0.31723, RAM Usage: 2.7GB, VRAM Usage: 7.0GB |
Validation loss: 0.25878, Validation acc: 90.57%
```

{"model_id":"8d1cadb2621941329e32c143b7686482","version_major":2,"version_minor":0}

{"model_id":"163441fbc6d9408d84cd542b2f97e837","version_major":2,"version_minor":0}

```
02: Train loss: 0.20265, RAM Usage: 2.7GB, VRAM Usage: 7.0GB |
Validation loss: 0.19366, Validation acc: 93.06%
```

{"model_id":"1dfc1b301cb64276932139103f59f437","version_major":2,"version_minor":0}

{"model_id":"6ec256f4954644c68d15f7cf9a05b5a2","version_major":2,"version_minor":0}

```
03: Train loss: 0.16588, RAM Usage: 2.7GB, VRAM Usage: 7.0GB |
Validation loss: 0.17980, Validation acc: 93.33%
```

{"model_id":"e791359e3ac54c7a9d2e82280679adbc","version_major":2,"version_minor":0}

{"model_id":"b1279329742c4a018a34bec2486d7d20","version_major":2,"version_minor":0}

```
04: Train loss: 0.14511, RAM Usage: 2.7GB, VRAM Usage: 7.0GB |
Validation loss: 0.17648, Validation acc: 93.60%
```

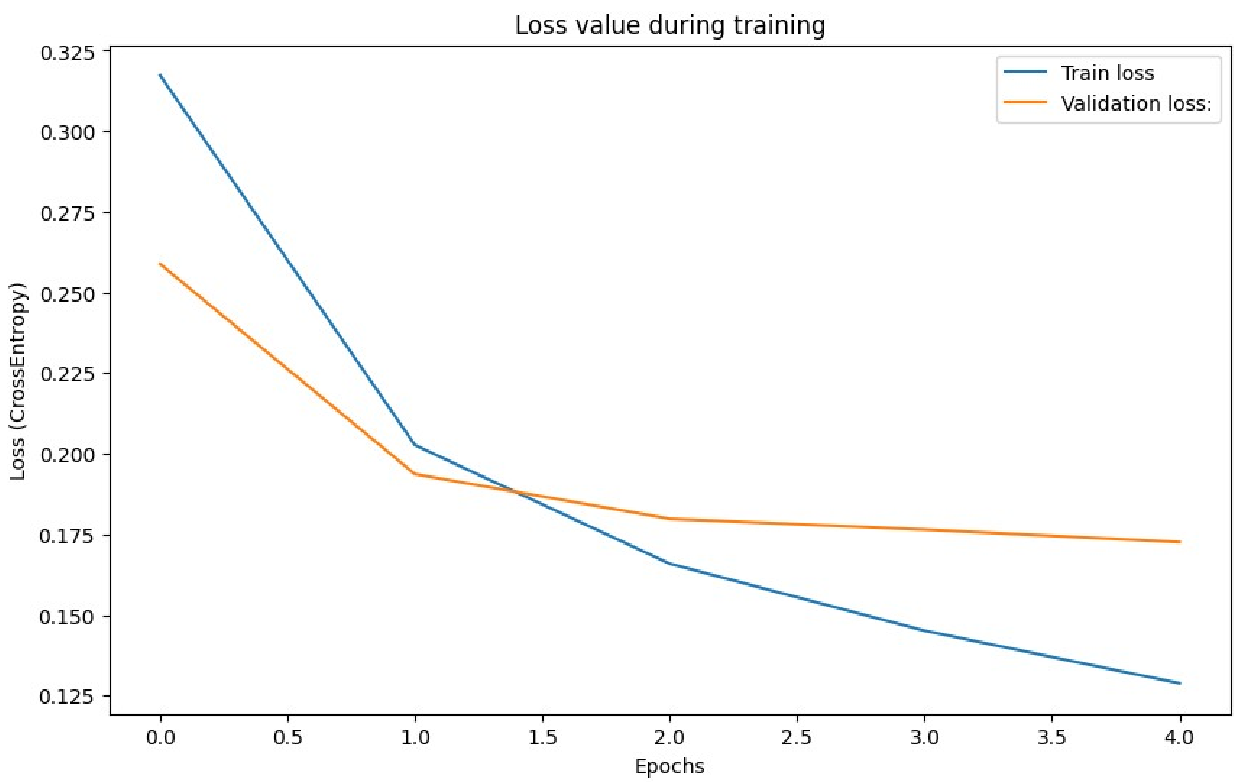
{"model_id":"4d1855143c0c43bc9b420f27e466e0e4","version_major":2,"version_minor":0}

{"model_id":"ace0d55e4c3a43008f66e64b1d40530e","version_major":2,"version_minor":0}

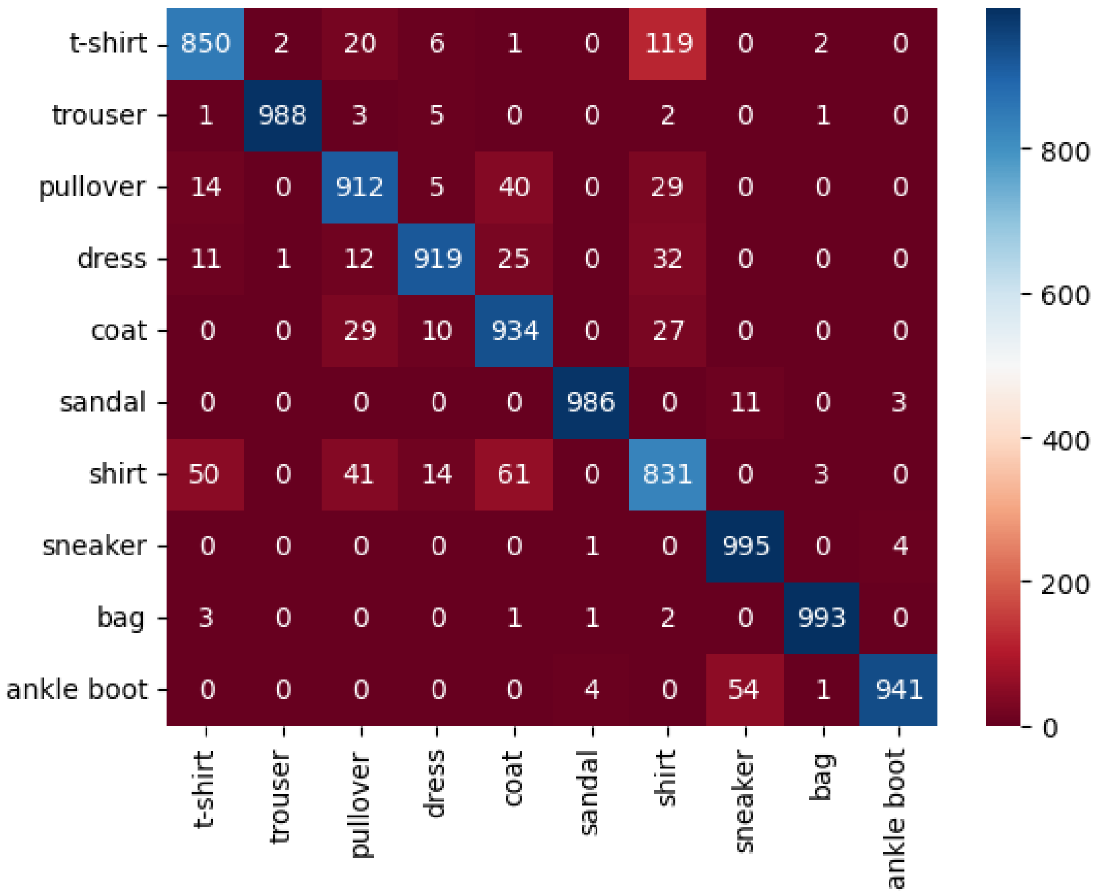
```
05: Train loss: 0.12883, RAM Usage: 2.7GB, VRAM Usage: 7.0GB |
Validation loss: 0.17260, Validation acc: 93.95%
```

{"model_id":"10ae27e8899b4efdbc75c63beacc987a","version_major":2,"version_minor":0}

```
              precision    recall  f1-score   support

           0       0.91      0.85      0.88      1000
           1       1.00      0.99      0.99      1000
           2       0.90      0.91      0.90      1000
           3       0.96      0.92      0.94      1000
           4       0.88      0.93      0.91      1000
           5       0.99      0.99      0.99      1000
           6       0.80      0.83      0.81      1000
           7       0.94      0.99      0.97      1000
           8       0.99      0.99      0.99      1000
           9       0.99      0.94      0.97      1000

    accuracy                           0.93     10000
   macro avg       0.94      0.93      0.94     10000
weighted avg       0.94      0.93      0.94     10000

Test loss: 0.1770
Time spent: 1281.24s
```




```
cmatrix = confusion_matrix(y_true, y_test_pred_rn34)
sns.heatmap(cmatrix, annot=True, fmt=".0f", cmap='RdBu',
xticklabels=labels_title, yticklabels=labels_title)
plt.show()
```



MobileNetV3

A MobileNetV3 é uma arquitetura de rede neural projetada para tarefas de visão computacional, especialmente otimizada para ambientes com recursos computacionais limitados, como dispositivos móveis.

> A arquitetura MobileNetV3 introduz o bloco invertido residual (MBConv), que é uma versão modificada do bloco residual usado em redes como a ResNet. Esse bloco ajuda a manter a eficiência computacional

```
class MyMobileNetV3(nn.Module):
    def __init__(self, in_channels=1, out_classes=10):
        super(MyMobileNetV3, self).__init__()
        self.mn3 = resnet34(weights=ResNet34_Weights.DEFAULT)
        self.mn3.conv1 = nn.Conv2d(1, 64, kernel_size=(7, 7),
stride=(2, 2), padding=(3, 3), bias=False)
        self.mn3.fc = nn.Linear(in_features=512, out_features=10,
bias=True)

    def forward(self, x):
        return self.mn3(x)
```

```python
    def predict(self, x):
        return torch.argmax(self.forward(x), dim=1)

output_size = 10

model_mn3 = MyMobileNetV3(1, output_size).to(device)
loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model_mn3.parameters(), lr=0.0001)

model_mn3
```

```
MyMobileNetV3(
  (mn3): ResNet(
    (conv1): Conv2d(1, 64, kernel_size=(7, 7), stride=(2, 2),
padding=(3, 3), bias=False)
    (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (relu): ReLU(inplace=True)
    (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1,
dilation=1, ceil_mode=False)
    (layer1): Sequential(
      (0): BasicBlock(
        (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
        (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
      (1): BasicBlock(
        (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
        (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
      (2): BasicBlock(
        (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
        (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1),
```

```
padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (layer2): Sequential(
      (0): BasicBlock(
        (conv1): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2),
padding=(1, 1), bias=False)
        (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (downsample): Sequential(
          (0): Conv2d(64, 128, kernel_size=(1, 1), stride=(2, 2),
bias=False)
          (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        )
      )
      (1): BasicBlock(
        (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
        (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
      (2): BasicBlock(
        (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
        (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
      (3): BasicBlock(
        (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
        (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
```

```
track_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (layer3): Sequential(
      (0): BasicBlock(
        (conv1): Conv2d(128, 256, kernel_size=(3, 3), stride=(2, 2),
padding=(1, 1), bias=False)
        (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (downsample): Sequential(
          (0): Conv2d(128, 256, kernel_size=(1, 1), stride=(2, 2),
bias=False)
          (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        )
      )
      (1): BasicBlock(
        (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
        (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
      (2): BasicBlock(
        (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
        (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
      (3): BasicBlock(
```

```
        (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
        (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
      (4): BasicBlock(
        (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
        (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
      (5): BasicBlock(
        (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
        (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (layer4): Sequential(
      (0): BasicBlock(
        (conv1): Conv2d(256, 512, kernel_size=(3, 3), stride=(2, 2),
padding=(1, 1), bias=False)
        (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (downsample): Sequential(
          (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2),
bias=False)
          (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
```

```
            )
         )
         (1): BasicBlock(
            (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
            (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
            (relu): ReLU(inplace=True)
            (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
            (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
         )
         (2): BasicBlock(
            (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
            (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
            (relu): ReLU(inplace=True)
            (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
            (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
         )
      )
      (avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
      (fc): Linear(in_features=512, out_features=10, bias=True)
   )
)
```

```
trtime_mn3, ram_mn3, vram_mn3 = train(model_mn3,
train_cnn_cons_loader, val_cnn_cons_loader, 5, loss_fn, optimizer)

y_test_pred_mn3, y_true, test_loss, prtime_mn3 = test(model_mn3,
test_cnn_cons_loader, loss_fn)
print(classification_report(y_true, y_test_pred_mn3))
print(f'Test loss: {test_loss:.4f}')
print(f'Time spent: {trtime_mn3:.2f}s')
models['MobileNetV3']['Accuracy'] = (accuracy_score(y_true,
y_test_pred_mn3))
models['MobileNetV3']['Tr Time'] = trtime_mn3
models['MobileNetV3']['Pr Time'] = prtime_mn3
models['MobileNetV3']['RAM'] = ram_mn3
models['MobileNetV3']['VRAM'] = vram_mn3
```

{"model_id":"4c6896db73d44d43871af159ce71ee93","version_major":2,"version_minor":0}

{"model_id":"a2d303f5992b4a17b1e2ed2c221f5d9e","version_major":2,"version_minor":0}

```
01: Train loss: 0.30794, RAM Usage: 2.7GB, VRAM Usage: 3.5GB |
Validation loss: 0.21168, Validation acc: 92.13%
```

{"model_id":"75668d0cdd424bc587f2bac882bb5a18","version_major":2,"version_minor":0}

{"model_id":"651e4071e47545d78585e24e39903ee3","version_major":2,"version_minor":0}

```
02: Train loss: 0.19648, RAM Usage: 2.7GB, VRAM Usage: 3.5GB |
Validation loss: 0.18851, Validation acc: 93.07%
```

{"model_id":"36fdadd31ef94e029bb8cc9a64010da0","version_major":2,"version_minor":0}

{"model_id":"58007deddb8d4445a518ffec89142a70","version_major":2,"version_minor":0}

```
03: Train loss: 0.16587, RAM Usage: 2.8GB, VRAM Usage: 3.5GB |
Validation loss: 0.17318, Validation acc: 93.65%
```

{"model_id":"7969d87f81c44972a76b9da9376e09f9","version_major":2,"version_minor":0}

{"model_id":"ef05c66e501a44fdb61de60df7743ec4","version_major":2,"version_minor":0}

```
04: Train loss: 0.14199, RAM Usage: 2.7GB, VRAM Usage: 3.5GB |
Validation loss: 0.17032, Validation acc: 93.78%
```

{"model_id":"e2c90ba1f610406e9442d058277e237f","version_major":2,"version_minor":0}

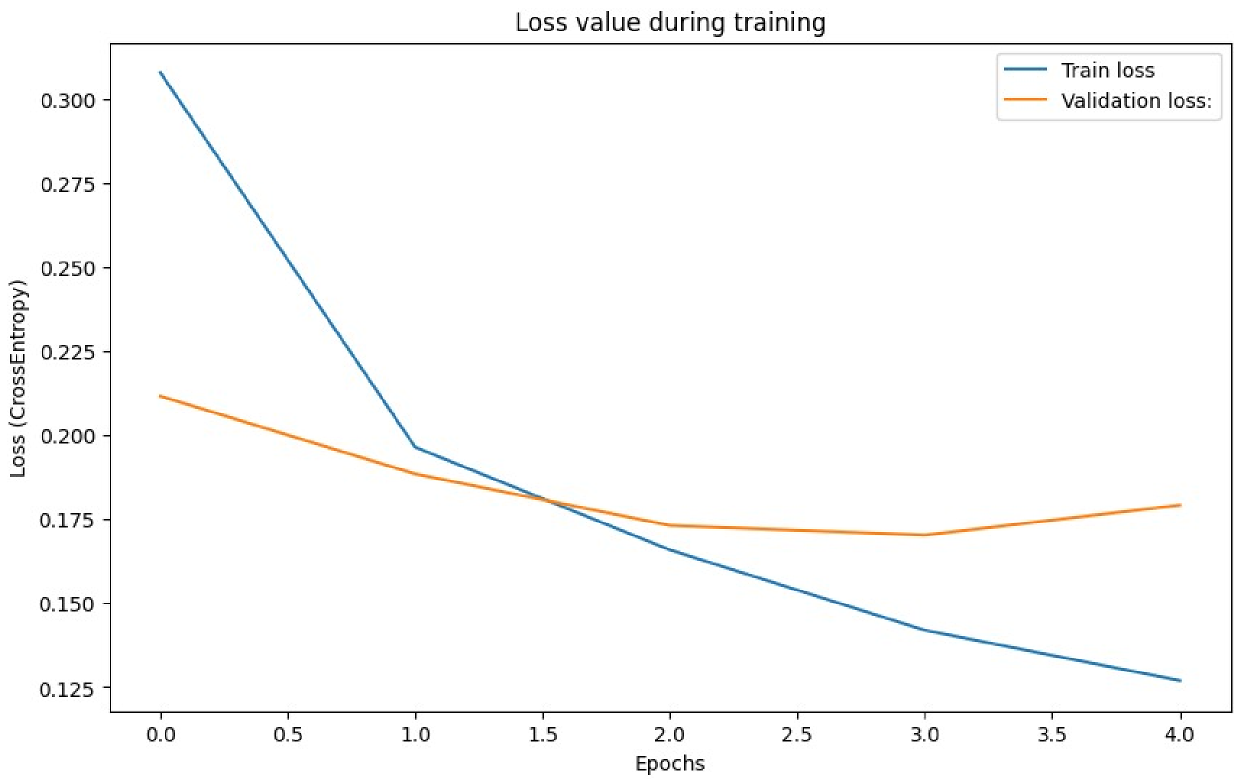
{"model_id":"29dff77cf2ff42318d782b0b760c784a","version_major":2,"version_minor":0}

```
05: Train loss: 0.12702, RAM Usage: 2.1GB, VRAM Usage: 3.5GB |
Validation loss: 0.17921, Validation acc: 93.50%
```

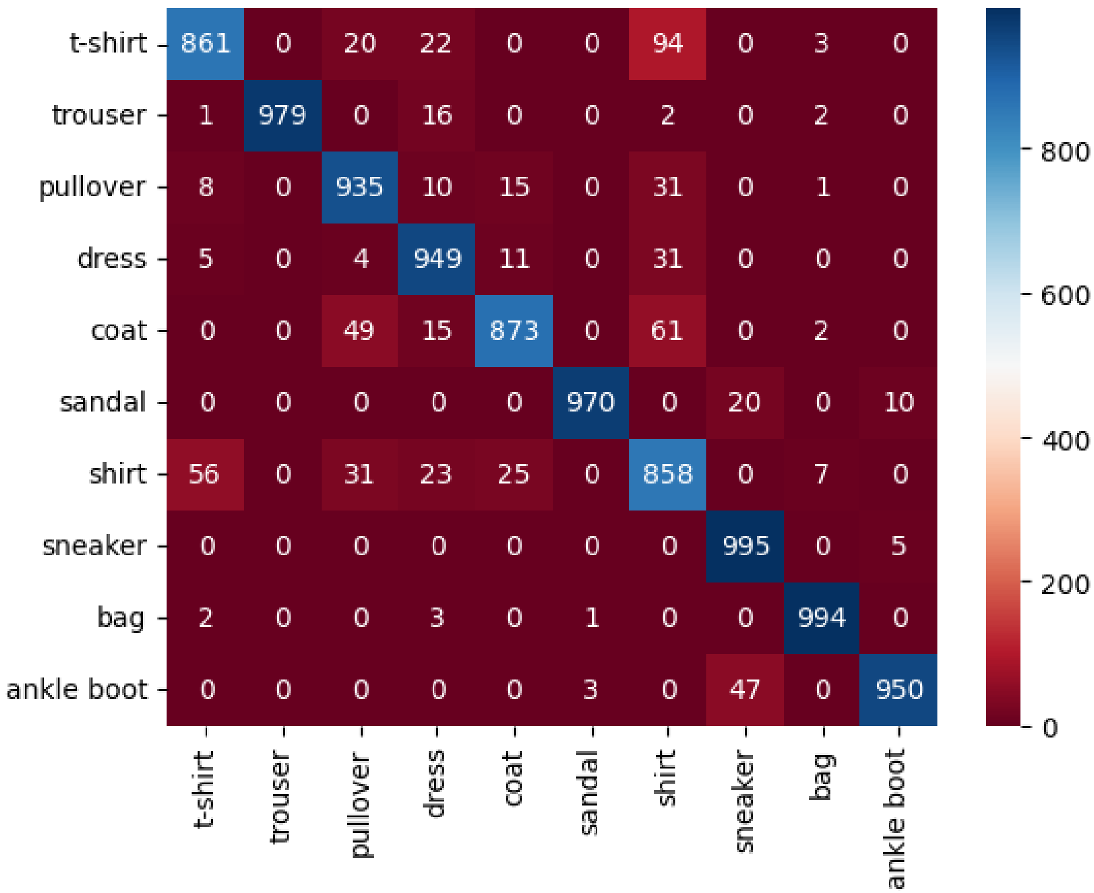
{"model_id":"a8e3db8a7d3d443e85adab104222daa5","version_major":2,"version_minor":0}

```
              precision    recall  f1-score   support

           0       0.92      0.86      0.89      1000
           1       1.00      0.98      0.99      1000
           2       0.90      0.94      0.92      1000
           3       0.91      0.95      0.93      1000
           4       0.94      0.87      0.91      1000
           5       1.00      0.97      0.98      1000
           6       0.80      0.86      0.83      1000
           7       0.94      0.99      0.97      1000
```

```
           8       0.99      0.99      0.99      1000
           9       0.98      0.95      0.97      1000

    accuracy                           0.94     10000
   macro avg       0.94      0.94      0.94     10000
weighted avg       0.94      0.94      0.94     10000

Test loss: 0.1756
Time spent: 1276.82s
```




```
cmatrix = confusion_matrix(y_true, y_test_pred_mn3)
sns.heatmap(cmatrix, annot=True, fmt=".0f", cmap='RdBu',
xticklabels=labels_title, yticklabels=labels_title)
plt.show()
```



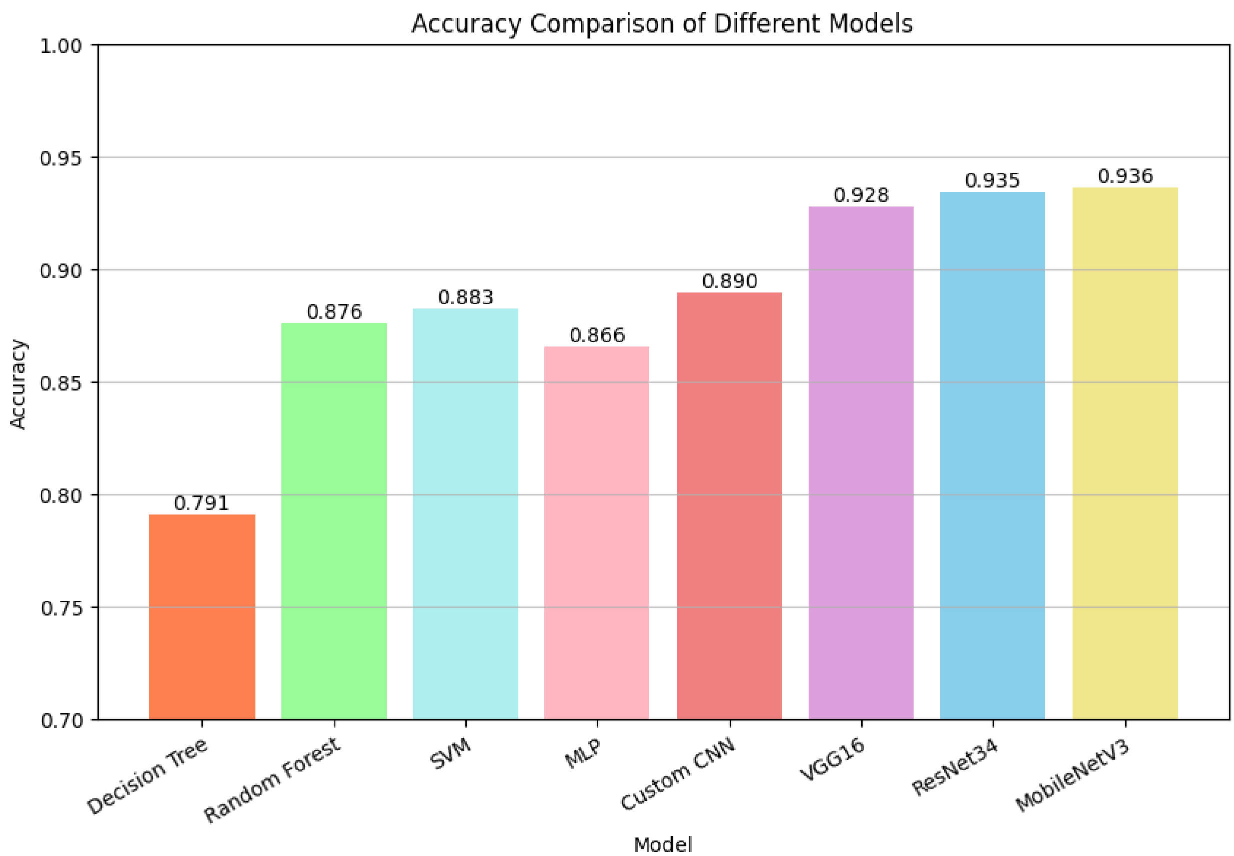
Comparação

Com todos os dados registrados, podemos comparar visualmente os modelos abordados. Nessa seção, levaremos em consideração as métricas de acurácia, tempo de execução e máximo uso de memória, todos coletados durante a execução acima.

> Também, resgataremos os relatórios de classificação para tentarmos ver quais classes dão mais problema para serem determinadas.

```
model_names = list(models.keys())
accuracies = [model['Accuracy'] for model in models.values()]
colors = plt.cm.viridis(np.linspace(0, 1, len(model_names)))
colors = [ 'coral', 'palegreen', 'paleturquoise', 'lightpink',
'lightcoral', 'plum', 'skyblue', 'khaki']

plt.figure(figsize=(10, 6))

acc_bars = plt.bar(model_names, accuracies, color=colors)
plt.title('Accuracy Comparison of Different Models')
```

```
plt.xlabel('Model')
plt.xticks(rotation=30, ha='right', rotation_mode='anchor')
plt.ylabel('Accuracy')
plt.grid(axis='y', linestyle='-', alpha=0.7)
plt.ylim(0.7, 1.0)

for bar, acc in zip(acc_bars, accuracies):
    plt.text(bar.get_x() + bar.get_width() / 2, bar.get_height() +
0.0, f'{acc:.3f}', ha='center', va='bottom')

plt.show()
```

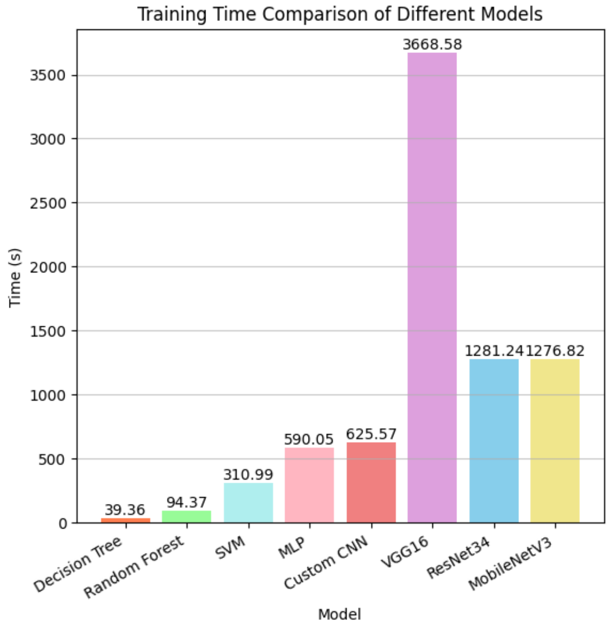


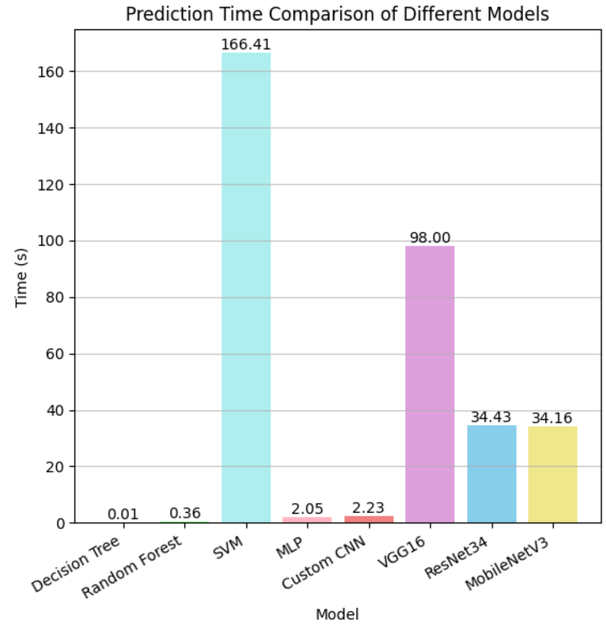
Pelo gráfico acima, é perceptível que as redes consolidadas (e pré treinadas) tiveram performance notavelmente superior, seguidas pela nossa rede convolucional própria. Isso evidencia a superioridade das convoluções na análise de imagens, assim como o impacto de um dataset diverso e volumoso (como no caso das redes consolidadas, treinadas no ImageNet).

> As relações de localidade e invariância presentes em imagens possibilitam as CNN de terem maior performance com menos custo.
>
> Os pesos pré treinados das CNNs consolidadas já aprenderam a extrair relações mais básicas e fundamentais, que podem ser reaproveitadas na nova aplicação. Bastando fazer o Fine Tuning.

Os melhores resultados foram da MobileNetV3 (seguido pela ResNet34 e VGG16), com valores acima de 90%.

```
model_names = list(models.keys())
trtimes = [model['Tr Time'] for model in models.values()]
prtimes = [model['Pr Time'] for model in models.values()]
colors = plt.cm.viridis(np.linspace(2, 3, len(model_names)))
colors = [ 'coral', 'palegreen', 'paleturquoise', 'lightpink',
'lightcoral', 'plum', 'skyblue', 'khaki']

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(14, 6))

trtime_bars = ax1.bar(model_names, trtimes, color=colors)
ax1.set_title('Training Time Comparison of Different Models')
ax1.set_xlabel('Model')
ax1.set_ylabel('Time (s)')

ax1.set_xticks(np.arange(len(model_names)))
ax1.set_xticklabels(model_names, rotation=30, ha='right')
ax1.grid(axis='y', linestyle='-', alpha=0.7)

for bar, time in zip(trtime_bars, trtimes):
    ax1.text(bar.get_x() + bar.get_width() / 2, bar.get_height() +
0.02, f'{time:.2f}', ha='center', va='bottom')

prtime_bars = ax2.bar(model_names, prtimes, color=colors)
ax2.set_title('Prediction Time Comparison of Different Models')
ax2.set_xlabel('Model')
ax2.set_ylabel('Time (s)')

ax2.set_xticks(np.arange(len(model_names)))
ax2.set_xticklabels(model_names, rotation=30, ha='right')
ax2.grid(axis='y', linestyle='-', alpha=0.7)

for bar, time in zip(prtime_bars, prtimes):
    plt.text(bar.get_x() + bar.get_width() / 2, bar.get_height() +
0.02, f'{time:.2f}', ha='center', va='bottom')

plt.show()
```





Acima estão presentes o tempo de execução (treinaento e predição) de cada modelo. Percebe-se que as redes consolidadas tiveram tempo de treinamento maior que as demais, provavelmente devido a sua maior robustez e profundidade, lidando com imagens maiores e passando por mais convoluções.

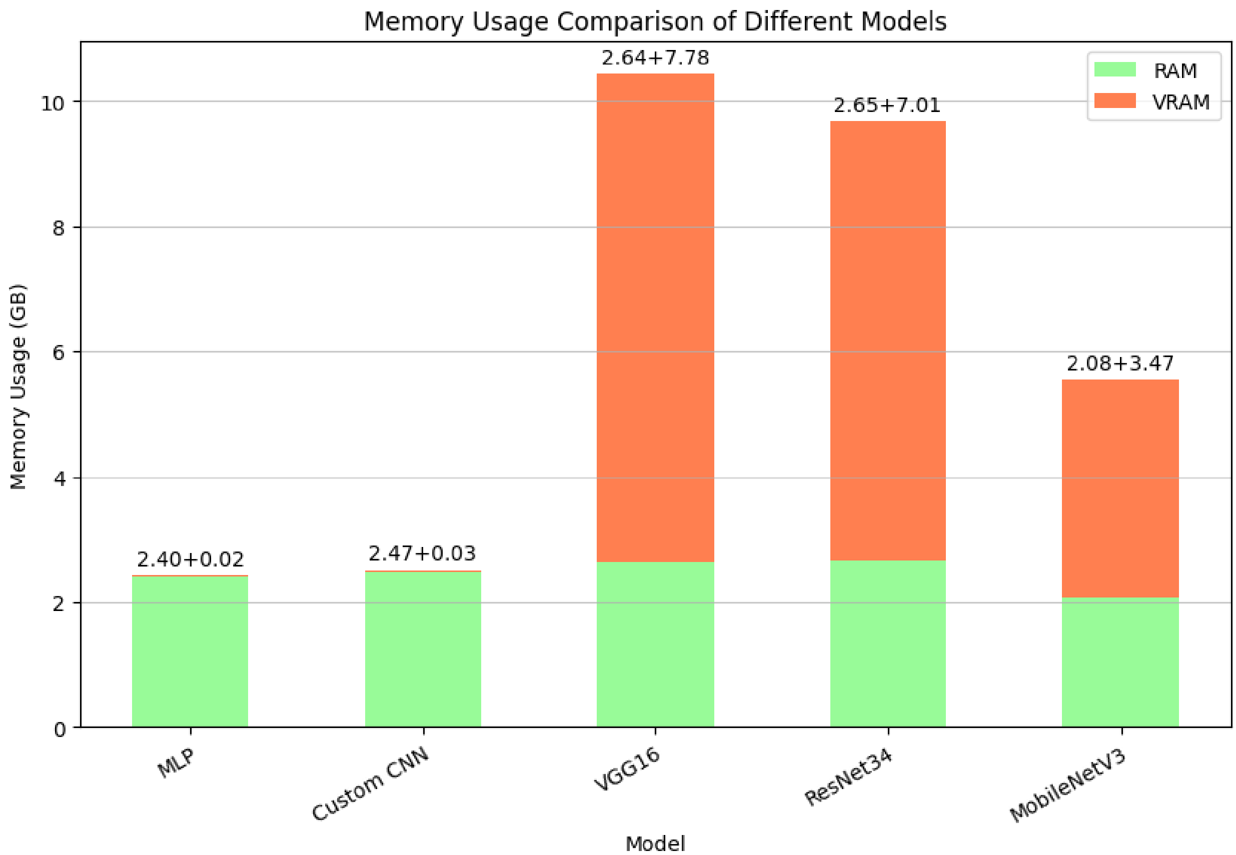
> Note a discrepância da VGG16, que demorou cerca de uma hora. Isso deve ter ocorrido porquê, além das camadas convolucionais, a rede tem uma etapa de MLP convencional, que necessita de mais parâmetros e cálculos (sem poder aproveitar os recursos da GPU).
>
> Redes idealizadas após a elaboração da Network in Network (NiN), como a ResNet e a MobileNet, tendem a não possuir camadas totalmente conectadas (MLP), acelerando sua execução e poupando custos.

Os modelos base tiveram tempos diversas vezes melhor, especialmente a Decision Tree, que operou em menos de um minuto.

Quando se trata do tempo de predição, o cenário é batente semelhante, com diferença da SVM, que apresentou um tempo varias vezes maior que os demais modelos.

```python
vram_models = {name: specs for name, specs in models.items() if specs['RAM'] > 0}
model_names = list(vram_models)
ram_values = [model['RAM'] for model in vram_models.values()]
vram_values = [model['VRAM'] for model in vram_models.values()]
colors = plt.cm.viridis(np.linspace(0, 1, len(model_names)))

bar_width = 0.50
bar_positions = np.arange(len(model_names))

plt.figure(figsize=(10, 6))
```

```python
plt.bar(bar_positions, ram_values, bar_width, label='RAM',
color='palegreen')
plt.bar(bar_positions, vram_values, bar_width, bottom=ram_values,
label='VRAM', color='coral')
plt.xlabel('Model')
plt.ylabel('Memory Usage (GB)')
plt.title('Memory Usage Comparison of Different Models')
plt.xticks(bar_positions, model_names, rotation=30, ha='right',
rotation_mode='anchor')
plt.grid(axis='y', linestyle='-', alpha=0.7)
plt.legend()

for idx, (ram_val, vram_val) in enumerate(zip(ram_values,
vram_values)):
    total_val = ram_val + vram_val
    plt.text(idx, total_val + 0.1, f'{ram_val:.2f}+{vram_val:.2f}',
ha='center', va='bottom')

plt.show()
```



Finalmente, temos o gráfico do uso de memória (RAM e VRAM). Percebe-se que o uso de RAM do sistema foi bem próximo em todas as aplicações (note que essa RAM é para o sistema todo, não apenas os modelos).

Já o uso de VRAM variou bastante. Enquanto a MLP (que não usa VRAM) e a CNN customizada usaram bem pouco, as redes consolidadas tiveram gastos potencialmente maiores. Um destaque ficou para a MobileNetV3, que gastou menos da metade que suas companheiras.

> A MobileNetV3 é pensada para rodar em dispositivos móveis, logo sua implementação foca em gastar menos recursos.
>
> Por usarmos funções próprias no `sklearn` para os modelos base, não foi possível registrar o uso de memória, mas foi visto que também ficou próximo dos 2 GB de RAM.
>
> Ao contrário do que geralmente se espera, a CNN customizada quase não usou VRAM. Porém, isso pode ser explicado pelo fato de usarmos as imagens pequenas originais (1 x 28 x 28), que não impactam muito na VRAM em apenas duas camadas convolucionais.

Performance por Classe

Voltando aos relatórios de classificação emitidos pelo `classification_report` e `confusion_matrix`, vemos que a classe com pior desempenho em todos os modelos foi a número 6 (shirt). Isso pode ser justificado pela similaridade dessa classe com outras (como pullover, coat e t-shirt).

Todos os modelos tiveram melhor performance nas classes 1 (trouser) e 8 (trouser), seguidas por 5 (sandal) e 9 (ankle boot). É perceptível como essas classes tem características mais próprias e definidas, ajudando a diferencia-las das demais.

> O desempenho por classe foi analisado de acordo com o precision, recall e f1-score.

Veredito

Pelos dados apontados acima, decidimos que a rede mais vantajosa foi a MobileNetV3, tendo maior acurácia que as demais, enquanto roda em um tempo razoável e usa menos RAM que as demais redes consolidadas. Caso se deseje um tempo ainda menor de execução e menos uso de memória, enquanto ainda mantém uma boa acurácia, a CNN customizada pode ser uma boa opção.

> Temos uma menção honrosa para a Random Forest, que teve uma acurácia basicamente igual à CNN enquanto rodou em um tempo mais que sete vezes menor.

Considerações Finais

As redes MLP e CNN customizada foram treinadas por 15 épocas e as redes CNN Consolidadas foram treinadas por 5 épocas. O método de treinamento implementado possui um recurso de earlystopping e esperar todas as redes atingirem a parada (sem fixar uma quantidade de épocas) poderia ser uma forma de comparar os resultados finais e, possivelmente, aumentar ligeiramente o desempenho dos modelos. Contudo, preferimos fixar os números de épocas para comparar os modelos por quantidades iguais de treinamento.

A arquitetura dos modelos MLP e CNN foram escolhidas para tentar usar a maioria dos recursos disponíveis (como batch normalization e dropout), e alguns testes de variações foram feitos anteriormente. As redes de melhor resultado foram mantidas. Poderíamos, por exemplo, ter aumentado a parcela convolucional da CNN, porem, em todos os testes, a performance for pior.