Lab 9:
Normalización de flujos

Júlia ORTEU

1 Enunciado

El dataset Olivetti Faces es un conjunto de datos que tiene 40 clases que corresponden a 10 fotografias de 40 personas. Son imagenes de 64x64, así que aprenderlas mediante un flujo directamente puede tardar mucho.

Una posibilidad es aplicar un método de reduccion de dimensionalidad, aprender el flujo en ese espacio y utilizarlo para muestrear nuevas caras simplemete aplicando el inverso del metodo de reduccion de dimensionalidad.

El conjunto de datos esta el `scikit-learn`, aplicaremos PCA como método de reduccion de dimensionalidad.

- Obten los datos del Olivetti Faces
- Aplica un PCA para obtener un nuevo conjunto de datos de dimensionalidad reducida donde la varianza explicada por los componentes principales sea alrededor del 90% (haz que sea una potencia de 2).
- Transforma los datos y crea con ellos un data loader
- Ajusta un flujo RealNVP como el que hemos hecho para los digitos
- Genera una muestra de los datos
- Calcula la log probabilidad de un subconjunto de los datos
- Haz una interpolacion entre dos de los ejemplos del conjunto de datos real para clases diferentes
- Haz un informe comentando lo que has hecho y explica las conclusiones que has sacado

Nota: No esperes que la generación sea perfecta, no uses un tamaño de batch muy grande (solo hay 400 ejemplos), ten cuidado tambien al seleccionar la tasa de aprendizaje su decaimiento. Ten paciencia, tarda un rato

2 Experimentación

2.1 Obtención de datos del Olivetti Faces

Para la obtención de datos, se utilizó la biblioteca de Python para cargar el conjunto de imágenes Olivetti Faces.

```
# Carreguem les dades de Olivietti faces
data = fetch_olivetti_faces()
X = data.images.reshape(data.images.shape[0], -1)  # (n_mostres, n_features)

# Visualitzar algunes mostres

n_mostres = 4

plt.figure(figsize=(10, 3))
for i in range(n_mostres):
    plt.subplot(1, n_mostres, i + 1)
    plt.imshow(data.images[np.random.randint(0, 400)], cmap='gray')
    plt.axis('off')
    plt.title(f'Mostra {i + 1}')

plt.show()
```

Listing 1: Código para la carga de los datos de Olivetti Faces

En la Figura 1 se muestra una visualización de cuatro muestras seleccionadas al azar del conjunto de datos Olivetti Faces. Esta representación gráfica brinda una primera impresión de la diversidad presente en las imágenes faciales que se utilizarán en las siguientes etapas del experimento.

Figure 1: Visualización de 4 muestras del dataset Olivetti Faces.

2.2 Aplicación de PCA para dimensionalidad reducida

Se aplicó el Análisis de Componentes Principales (PCA) para obtener un nuevo conjunto de datos con una dimensionalidad reducida, ajustando la varianza explicada por los componentes principales a alrededor del 90%.

```
# Apliquem una PCA para fer una reducció de dimensionalitat

pca = PCA(n_components=0.90, svd_solver='full')
X_pca = pca.fit_transform(X)
n_components = pca.n_components_

print('Nombre de components: ', n_components)
```

Listing 2: Aplicación PCA

```
Output: Nombre de components PCA =  66
```

En la Figura 2, se muestra un gráfico de la acumulación de la varianza explicada por las dimensiones iniciales resultantes.

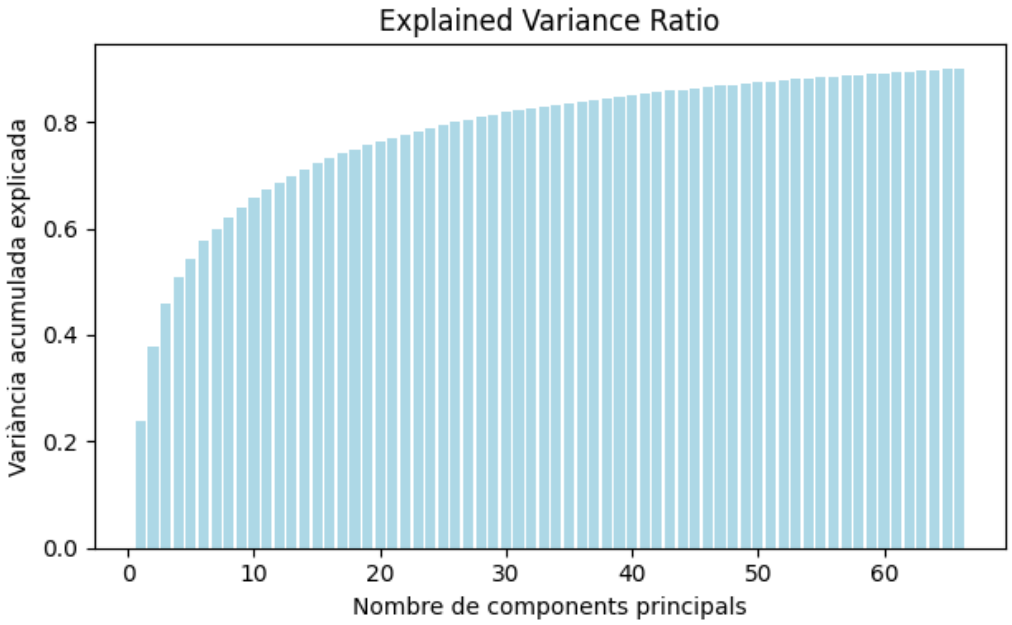

Figure 2: *Initial Explained Variance Ratio Plot*

A continuación, se aseguró que el número de dimensiones fuera una potencia de dos.

```
# Ha de ser una potencia de dos per tant:
closest_power_of_2 = 2**round(np.log2(n_components))

print('ºN components més propera èpotncia de dos = ', closest_power_of_2)

pca = PCA(n_components=closest_power_of_2)
X_pca = pca.fit_transform(X)
```

Listing 3: PCA con dimensiones potencia de dos

```
Output: Nº components més propera potència de dos = 64
```

Esto resultó en el gráfico de la explicación de la varianza por dimensiones potencia de dos mostrado en la Figura 3 con 64 componentes.

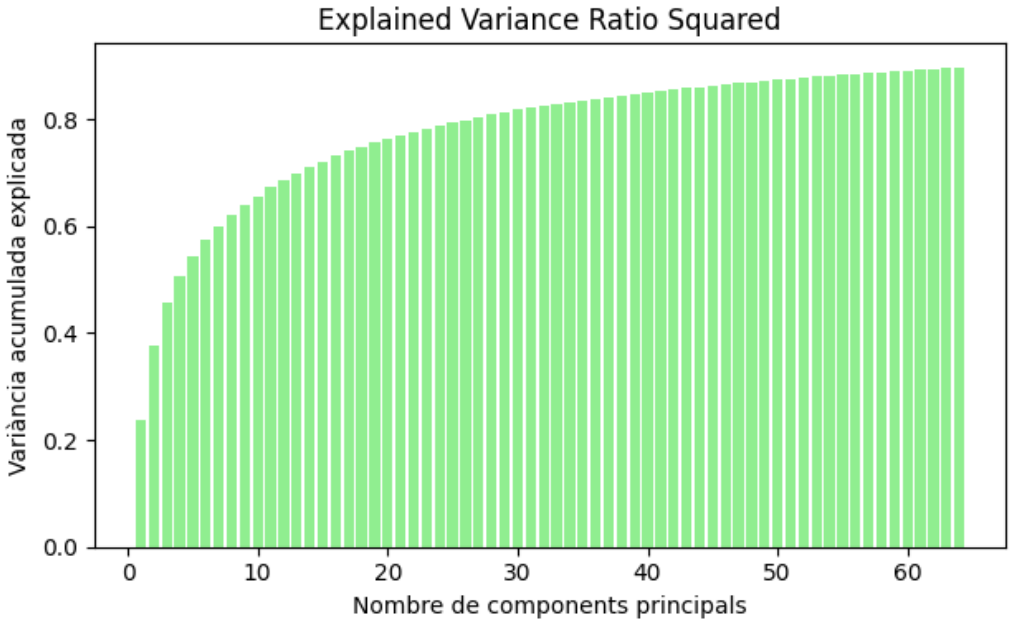

Figure 3: *Final Explained Variance Ratio Plot*

2.3 Transformación de datos y creación de DataLoader

Los datos fueron transformados según el resultado del PCA y se creó un DataLoader para facilitar el manejo de los datos durante el proceso con `batch_size = 32`.

```
# Creem un dataloader
tensor_x = torch.tensor(X_pca).float()
dataset = TensorDataset(tensor_x)
dataloader = DataLoader(dataset, batch_size=32, shuffle=True)
```

Listing 4: Creación de un `DataLoader`.

2.4 Ajuste de un flujo RealNVP

Se ajustó un flujo RealNVP al conjunto de datos, siguiendo un procedimiento similar al utilizado para los dígitos en los experimentos anteriores presentados por el profesor en la sesión de laboratorio.

```
# Definició del flux RealNVP
K = 8
latent_size = closest_power_of_2
b = torch.Tensor([1 if i % 2 == 0 else 0 for i in range(latent_size)]).to(device)
flows = []

for i in range(K):
    s = nf.nets.MLP([latent_size, 2 * latent_size, 2 * latent_size, latent_size],
        init_zeros=True)
    t = nf.nets.MLP([latent_size, 2 * latent_size, 2 * latent_size, latent_size],
        init_zeros=True)
    if i % 2 == 0:
        flows += [nf.flows.MaskedAffineFlow(b, t, s)]
    else:
        flows += [nf.flows.MaskedAffineFlow(1 - b, t, s)]
    flows += [nf.flows.ActNorm(latent_size)]

base = nf.distributions.DiagGaussian(latent_size)
rnvp = nf.NormalizingFlow(base, flows).to(device)

# Bucle d'entrenament
def train_loop(model, optimizer, scheduler, dataloader, epochs, conditional=False):
    hist_loss = []
    pbar = tqdm(range(epochs))
    for epoch in pbar:
        running_loss = 0.0
        for i, (data,) in enumerate(dataloader):
            optimizer.zero_grad()
            data = data.to(device)
            # Calcula la èprdua depenent de si és condicional o no
            if not conditional:
                loss = model.forward_kld(data)
            # Si la èprdua no és nan o inf, fa el backpropagation
            if ~(torch.isnan(loss) | torch.isinf(loss)):
                loss.backward()
                optimizer.step()
            running_loss += loss.item()

         if scheduler is not None:
            scheduler.step()
            lr = f'lr: {scheduler.get_last_lr()[0]:.4E}'
        else:
            lr = ''

        hist_loss.append(running_loss / i)
        pbar.set_description(f'loss: {running_loss / i:3.4f}:{lr}')

    return hist_loss
```

Listing 5: Configuración y definición del flujo RealNVP

La función `train_loop` se encarga del entrenamiento del modelo de flujo RealNVP a lo largo de múltiples épocas. El bucle de entrenamiento incluye la optimización del modelo mediante el descenso de gradiente utilizando el algoritmo Adam. Además, se utiliza un programador de tasas de aprendizaje (`lr_scheduler`) para ajustar dinámicamente la tasa de aprendizaje durante el entrenamiento.

El número total de épocas se establece en 150. El historial de pérdida (`hist_loss`) se actualiza en cada época y se devuelve al final de la función.

```
# Iniciem i entrenem
optimizer = torch.optim.Adam(rnvp.parameters(), lr=1e-4)
scheduler = lr_scheduler.StepLR(optimizer, step_size=12, gamma=0.6)
epochs = 150

hist_loss = train_loop(rnvp, optimizer, scheduler, dataloader, epochs)
```

Listing 6: Inciamos y entrenamos el modelo.

```
Output: loss: -5.7816:lr: 2.1768E-07: 100%|     | 150/150 [00:43<00:00,  3.44it/s]
```

La Figura 4 muestra un histograma de la pérdida durante el entrenamiento del flujo RealNVP.

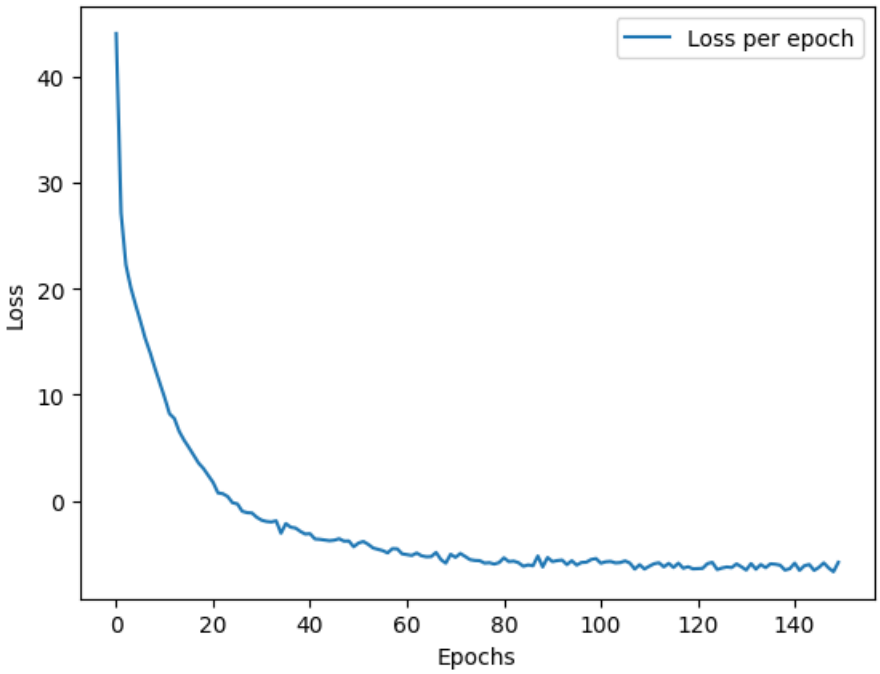

Figure 4: Histograma de la Loss durante el entrenamiento

2.5 Generación de muestra de datos

Se generó una muestra de datos utilizando el flujo RealNVP entrenado.

```
# Generem mostres i les visualitzem
nim = 3
samples, _ = lrnvp.sample(num_samples=nim * nim)
samples = samples.cpu().detach().numpy()

# Reescalem les mostres utilitzant un PCA invers
fig = plt.figure(figsize=(6, 6))
for i in range(nim * nim):
    plt.subplot(nim, nim, i + 1)
    sample_img = pca.inverse_transform(samples[i]).reshape(64, 64)
    plt.imshow(sample_img, cmap='gray')
    plt.axis('off')
```

Listing 7: Generación de una muestra de datos.

La Figura 5 muestra nueve muestras generadas por el modelo entrenado mediante el flujo RealNVP. Estas muestras proporcionan una visión visual de la capacidad del modelo para generar datos similares al conjunto de entrenamiento original.

Figure 5: Nueve muestras generadas por el modelo.

2.6 Cálculo de log probabilidad

Se calculó la log probabilidad del subconjunto de las muestras anteriores generadas por el modelo entrenado mediante el flujo RealNVP. El siguiente código muestra cómo se realizó este cálculo:

```
# Calcul de la log prbabilitat de les mostres generades
subset_samples = samples[:16]
subset_samples = torch.tensor(subset_samples, device=device)
log_prob_subset = lrnvp.log_prob(subset_samples)

print("Log Probabilitat del Subconjunt:")
print(log_prob_subset)
```

Listing 8: Cálculo de la log probabilidad de un subconjunto de muestras generadas.

```
Output: Log Probabilitat del Subconjunt:
    tensor([-21.8976, -24.9662, -31.1674,  19.9765, -15.3981,  -8.9472, -23.6099,
              2.6014,  -7.9552], device='cuda:0', grad_fn=<AddBackward0>)
```

Estos valores representan la log probabilidad asociada a cada muestra generada por el modelo en el subconjunto seleccionado. Pueden utilizarse para evaluar la calidad de las muestras generadas en comparación con el conjunto de datos original.

2.7 Interpolación entre ejemplos de clases diferentes

Se realizaron dos interpolaciones entre 4 ejemplos del conjunto de datos real, pertenecientes a clases diferentes, con el objetivo de explorar las capacidades del modelo en la generación de transiciones suaves.

```
# Interpolació entre dues mostres escollides

start = lrnvp.inverse(torch.tensor(samples[0]).to(device))[0]
end = lrnvp.inverse(torch.tensor(samples[3]).to(device))[0]

interp = []
for v in torch.arange(0, 1, 0.1):
    interp.append(torch.lerp(start, end, v.to(device)))
fig = plt.figure(figsize=(15, 10))

for i, v in enumerate(interp):
    interp_img = pca.inverse_transform(lrnvp(v.unsqueeze(0))[0].cpu().detach().numpy
        ()).reshape(64, 64)  # PCA invers
    plt.subplot(1, len(interp), i + 1)
    plt.imshow(interp_img, cmap='gray')
    plt.axis('off')
```

Listing 9: Interpolación entre dos muestras generadas.

Figure 6: Primera interpolación entre ejemplos de clases diferentes.

Figure 7: Segunda interpolación entre ejemplos de clases diferentes.

Las Figuras 6 y 7 muestran los resultados de las interpolaciones. Cada par de imágenes representa el proceso continuo de transición entre dos ejemplos de clases diferentes. Estas visualizaciones proporcionan una representación gráfica de la capacidad del modelo para generar transiciones suaves entre distintas clases del conjunto de datos.

3 Conclusiones

En este experimento con el flujo RealNVP aplicado a Olivetti Faces:

- Se redujo la dimensionalidad de los datso con PCA.
- El modelo generó nuevas muestras con éxito.
- Interpolaciones suaves entre clases demostraron coherencia.
- Log probabilidad evaluada para métricas cuantitativas.
- Las muestras nuevas presentan formas parecidas a caras, aunque la mayoria de los resultados están lejos de una generación perfecta.

En resumen, el modelo muestra habilidades prometedoras, generando transiciones suaves y variadas en el espacio latente del conjunto de datos. Sin embargo, la generación perfecta aún representa un desafío