Representation Learning

Akin Wilson

sky glass

Introduction

Representation learning has many applications in the domain of DL. For example, a pracitioner if needed could used a overcomplete sparse autoencoder to find a memory efficient represenation of data. Another application is as a means of feature extraction upstream before being passed to a memory-constraint classifier.

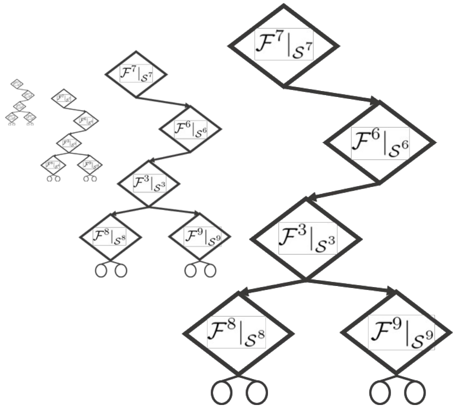
Weights, states and the ML model

A machine learning model is built using a dataset, an architecture and an optimizer

Weights, states and the ML model

A machine learning model is built using a dataset, an architecture and an optimizer

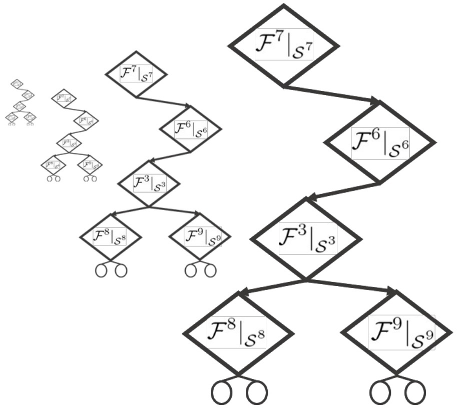
Architecture: Light gradient boosting ensemble



Weights, states and the ML model

A machine learning model is built using a dataset, an architecture and an optimizer

Architecture: Light gradient boosting ensemble



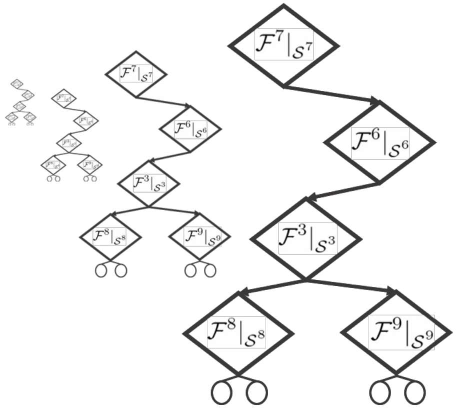
Dataset: unsupervised dataset sampled
from data-generating distribution

$$X = \begin{bmatrix} 1 & 0 & 1 & 0.56 & 1 & 0.3 \\ 1 & 0 & 0 & 0.32 & 0 & 0.9 \\ 1 & 1 & 1 & 0.99 & 1 & 0.1 \\ 0 & 0 & 0 & 0.12 & 1 & 0.23 \\ 1 & 1 & 1 & 0.62 & 1 & 0.1 \\ 1 & 1 & 0 & 0.40 & 1 & 0.43 \end{bmatrix}$$

Weights, states and the ML model

A machine learning model is built using a dataset, an architecture and an optimizer

Architecture: Light gradient boosting ensemble



Dataset: unsupervised dataset sampled from data-generating distribution

$$X = \begin{bmatrix} 1 & 0 & 1 & 0.56 & 1 & 0.3 \\ 1 & 0 & 0 & 0.32 & 0 & 0.9 \\ 1 & 1 & 1 & 0.99 & 1 & 0.1 \\ 0 & 0 & 0 & 0.12 & 1 & 0.23 \\ 1 & 1 & 1 & 0.62 & 1 & 0.1 \\ 1 & 1 & 0 & 0.40 & 1 & 0.43 \end{bmatrix}$$

Optimizer: Gradient boosting

$$\hat{F} = \arg\min_{F} \mathbb{E}_{x,y}[L(y, F(x))].$$

$$\hat{F}(x) = \sum_{m=1}^{M} \gamma_m h_m(x) + \text{const}.$$

$$F_0(x) = \arg\min_{\gamma} \sum_{i=1}^{n} L(y_i, \gamma),$$

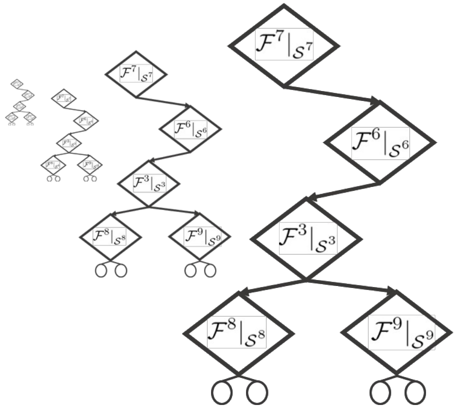
$$F_m(x) = F_{m-1}(x) + \arg\min_{h_m \in \mathcal{H}} \left[\sum_{i=1}^{n} L(y_i, F_{m-1}(x_i) + h_m(x_i)) \right],$$

$$F_m(x) = F_{m-1}(x) - \gamma \sum_{i=1}^{n} \nabla_{F_{m-1}} L(y_i, F_{m-1}(x_i))$$

Weights, states and the ML model

A machine learning model is built using a dataset, an architecture and an optimizer

Architecture: Light gradient boosting ensemble



Dataset: unsupervised dataset sampled from data-generating distribution

$$X = \begin{bmatrix} 1 & 0 & 1 & 0.56 & 1 & 0.3 \\ 1 & 0 & 0 & 0.32 & 0 & 0.9 \\ 1 & 1 & 1 & 0.99 & 1 & 0.1 \\ 0 & 0 & 0 & 0.12 & 1 & 0.23 \\ 1 & 1 & 1 & 0.62 & 1 & 0.1 \\ 1 & 1 & 0 & 0.40 & 1 & 0.43 \end{bmatrix}$$

Optimizer: Gradient boosting

$$\hat{F} = \arg\min_{F} \mathbb{E}_{x,y}[L(y, F(x))].$$

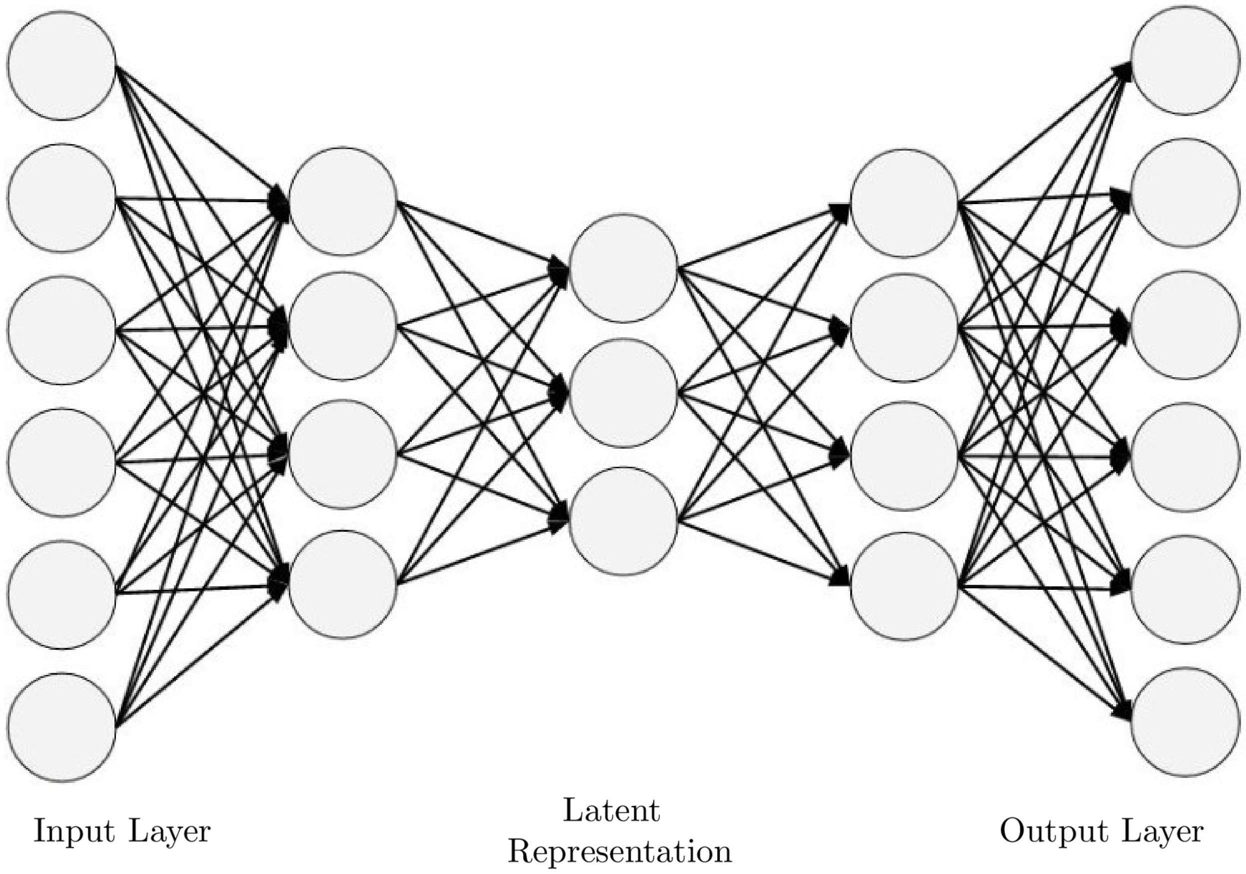
$$\hat{F}(x) = \sum_{m=1}^{M} \gamma_m h_m(x) + \text{const}.$$

$$F_0(x) = \arg\min_{\gamma} \sum_{i=1}^{n} L(y_i, \gamma),$$

$$F_m(x) = F_{m-1}(x) + \arg\min_{h_m \in \mathcal{H}} \left[\sum_{i=1}^{n} L(y_i, F_{m-1}(x_i) + h_m(x_i)) \right],$$

$$F_m(x) = F_{m-1}(x) - \gamma \sum_{i=1}^{n} \nabla_{F_{m-1}} L(y_i, F_{m-1}(x_i))$$

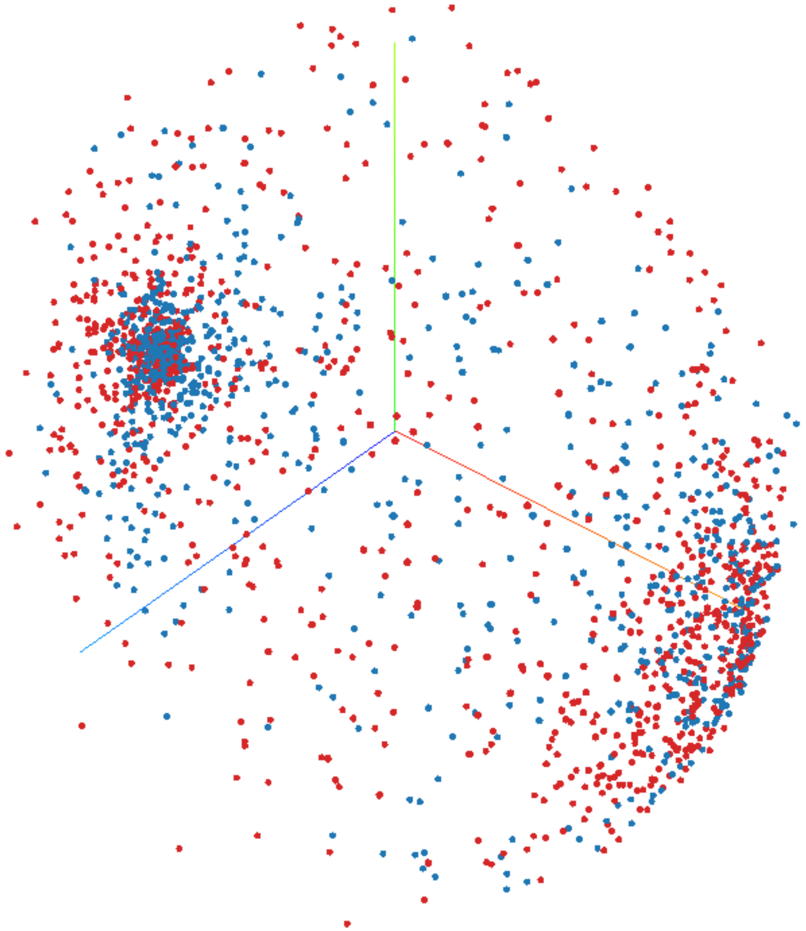
The Autoencoder



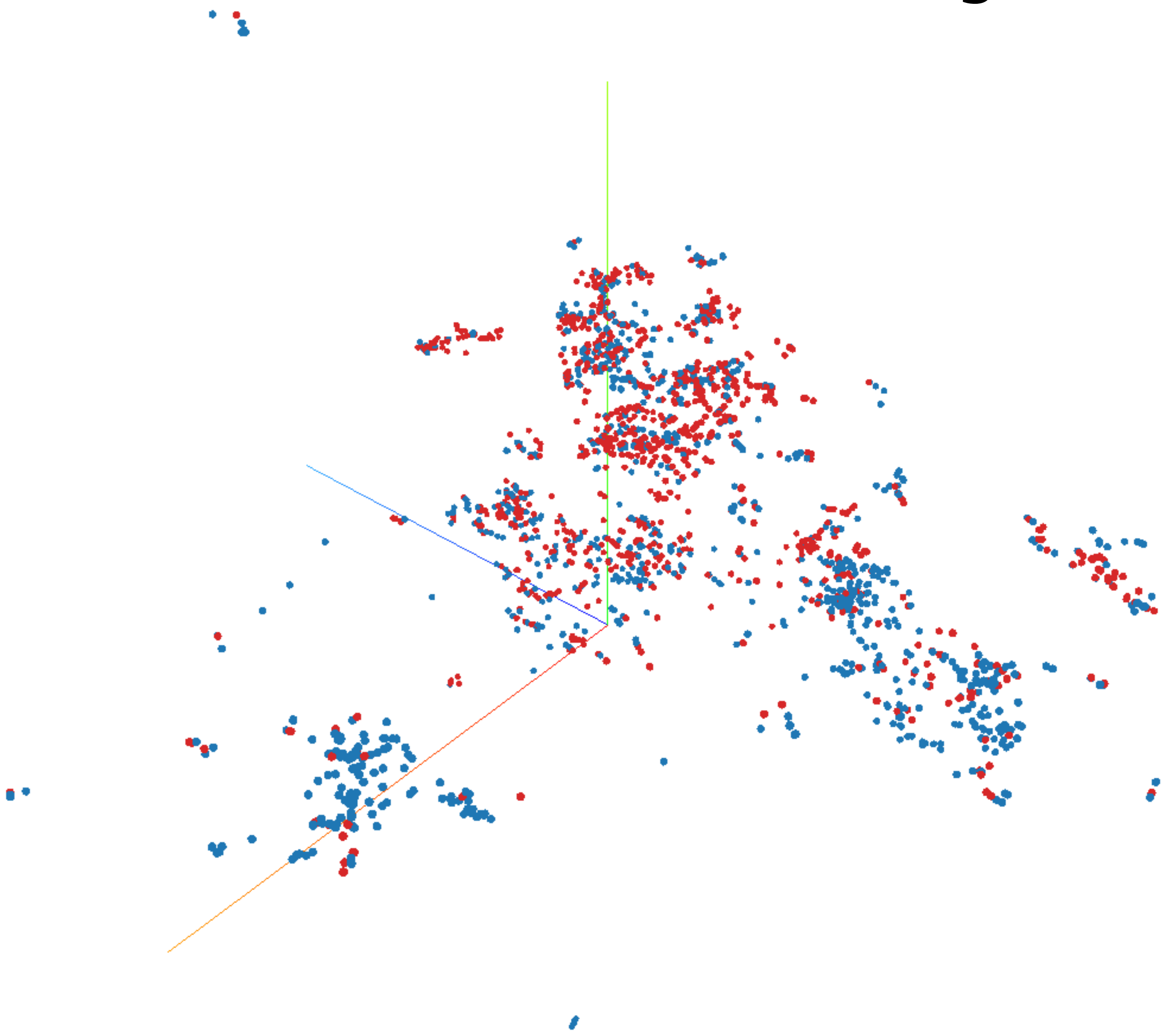
Undercomplete variant

```python
class CNNAE(nn.Module):
    def __init__(self, n_input=1, latent_dim=1024, stride=16, n_channel=32):
        super().__init__()
        self.device = "cuda" if torch.cuda.is_available() else "cpu"
        self.n_channel = n_channel
        # encoder layers
        self.e_conv1 = nn.Conv1d(n_input, n_channel, kernel_size=80, stride=stride)
        self.e_bn1 = nn.BatchNorm1d(n_channel)
        self.e_pool1 = nn.MaxPool1d(4, return_indices=True)
        self.e_conv2 = nn.Conv1d(n_channel, n_channel, kernel_size=3)
        self.e_bn2 = nn.BatchNorm1d(n_channel)
        self.e_pool2 = nn.MaxPool1d(4, return_indices=True)
        self.e_conv3 = nn.Conv1d(n_channel, 2 * n_channel, kernel_size=3)
        self.e_bn3 = nn.BatchNorm1d(2 * n_channel)
        self.e_pool3 = nn.MaxPool1d(4, return_indices=True)
        self.e_conv4 = nn.Conv1d(2 * n_channel, 2 * n_channel, kernel_size=3)
        self.e_bn4 = nn.BatchNorm1d(2 * n_channel)
        self.e_pool4 = nn.MaxPool1d(2, return_indices=True)
        self.e_fc4 = nn.Linear(2 * n_channel * 28, latent_dim)
        # decoder layers
        self.d_fc4 = nn.Linear(latent_dim, 2 * n_channel * 28)
        self.d_pool4 = nn.MaxUnpool1d(2)
        self.d_bn4 = nn.BatchNorm1d(2 * n_channel)
        self.d_conv4 = nn.ConvTranspose1d(2 * n_channel, 2 * n_channel, kernel_size=3)
        self.d_pool3 = nn.MaxUnpool1d(4)
        self.d_bn3 = nn.BatchNorm1d(2 * n_channel)
        self.d_conv3 = nn.ConvTranspose1d(2 * n_channel, n_channel, kernel_size=3)
        self.d_pool2 = nn.MaxUnpool1d(4)
        self.d_bn2 = nn.BatchNorm1d(n_channel)
        self.d_conv2 = nn.ConvTranspose1d(n_channel, n_channel, kernel_size=3)
        self.d_pool1 = nn.MaxUnpool1d(4)
        self.d_bn1 = nn.BatchNorm1d(n_channel)
        self.d_conv1 = nn.ConvTranspose1d(n_channel, n_input, kernel_size=80, stride=stride)


    def encode(self, x):
        x = self.e_conv1(x)
        x = F.relu(self.e_bn1(x))
        x, idx1 = self.e_pool1(x)
        x = self.e_conv2(x)
        x = F.relu(self.e_bn2(x))
        x, idx2 = self.e_pool2(x)
        x = self.e_conv3(x)
        x = F.relu(self.e_bn3(x))
        x, idx3  = self.e_pool3(x)
        x = self.e_conv4(x)
        x = F.relu(self.e_bn4(x))
        x = x.view(x.shape[0], -1)
        x = self.e_fc4(x)
        return idx1, idx2, idx3, x


    def decode(self, idx1, idx2, idx3, x):
        bs = x.shape[0]
        x = self.d_fc4(x)
        x = x.view(bs, 2 * self.n_channel,  28)
        x = F.relu(self.d_bn4(x))
        x = self.d_conv4(x)
        x = self.d_pool3(x, idx3)
        x = F.relu(self.d_bn3(x))
        x = self.d_conv3(x)
        padding = idx2.shape[2] - x.shape[2]
        pad = torch.zeros((bs,32, padding),device=self.device)
        x = torch.cat([x,pad],dim=2)
        x = self.d_pool2(x, idx2)
        x = F.relu(self.d_bn2(x))
        x = self.d_conv2(x)
        padding = idx1.shape[2] - x.shape[2]
        pad = torch.zeros((bs,32, padding), device=self.device)
        x = torch.cat([x,pad],dim=2)
        x = self.d_pool1(x, idx1)
        x = F.relu(self.d_bn1(x))
        x = self.d_conv1(x)
        return x


    def forward(self, x):
        idx1, idx2, idx3, encoded_x = self.encode(x)
        decoded_x = self.decode(idx1, idx2, idx3,encoded_x)
        return decoded_x
```

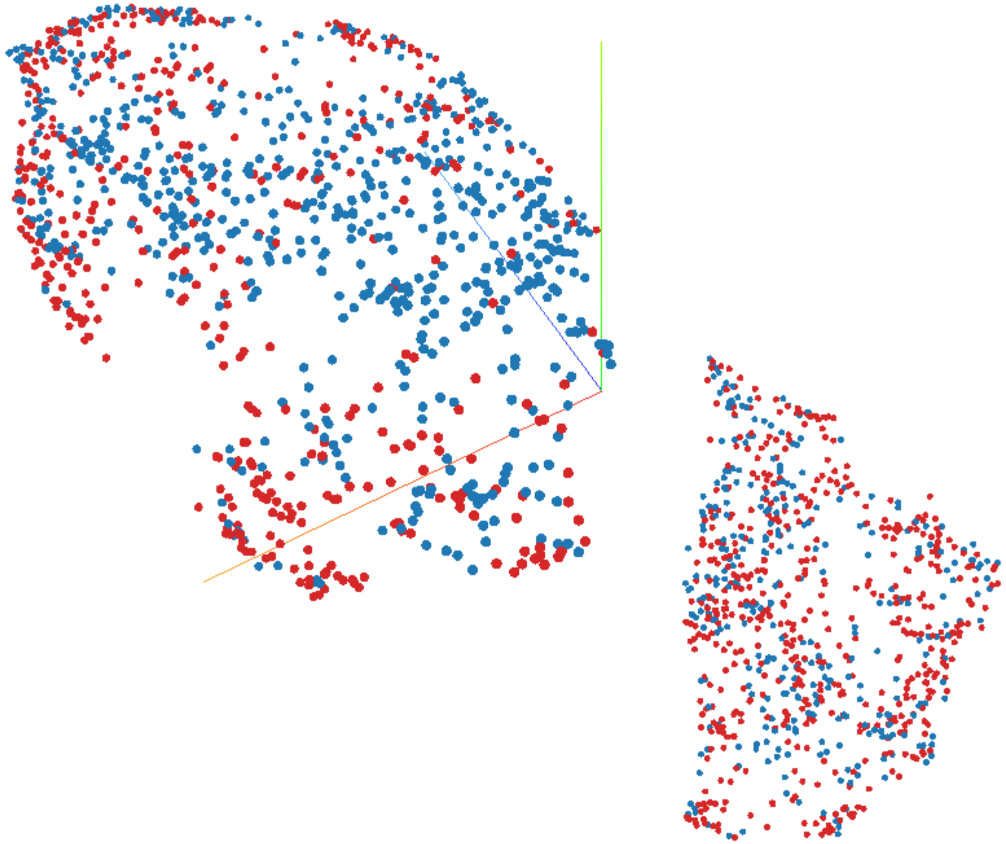
PCA: Principle component analysis



T-SNE :t-distributed stochastic neighbor embedding



UMAP: Uniform manifold approximation and projection



Downstream latent code classification

```
class AE_classifier(nn.Module):
    '''
    Classifier takes latent code and performs predictions using the latent code

    Applications:
        Upstream feature extraction for memory-constraint classifier
    '''


    def __init__(self,latent_dim, compression_factor=3):
        '''
        Compression factor detemines the intermitten dimension reduction factor of the dense network. If latent dim 300, then dense layer 1 out will be 100, and then 33 and finally 1.
        '''
        super().__init__()
        self.latent_dim = latent_dim

        dense_layer_1_output = int(latent_dim / compression_factor)
        dense_layer_2_output = int(dense_layer_1_output / compression_factor)

        module_dict = {
            "DenseLayer1": nn.Linear(latent_dim, dense_layer_1_output),
            "DenseLayer2": nn.Linear(dense_layer_1_output, dense_layer_2_output),
            "DenseLayer3": nn.Linear(dense_layer_2_output, 1),
        }

        self.layers =  torch.ModuleDict(module_dict)

    def forward(self, x):
        logits = self.layers(x)
        return logits
```

Applications

Applications

Applications

From discrimantive to generative

Synthesis

Synthesis

Synethically augmented fitting results

Synethically augmented fitting results