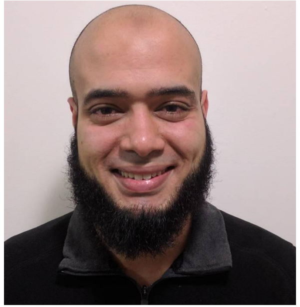
Machine Learning

Network Design Challenge

Mostafa S. Ibrahim
Teaching, Training and Coaching for more than a decade!

Artificial Intelligence & Computer Vision Researcher
PhD from Simon Fraser University - Canada
Bachelor / MSc from Cairo University - Egypt
Ex-(Software Engineer / ICPC World Finalist)



Regression with missing values

- Assume we have a source of Data named X1. Each example of this source has **40** numerical features in range [0-1]
    - There are is relevant semantic connection between every pair of features (in this source)
    - Assume we have another source of Data named X2. Each example of this source has **60** numerical features in range [0-1]
- We can concatenate an exame x = [x1 + x2] of **100 features** and use it to regress a value y
- The challenge: While all our data has no missing features, in real inference, up to 90% of the features can go missing.

Challenge

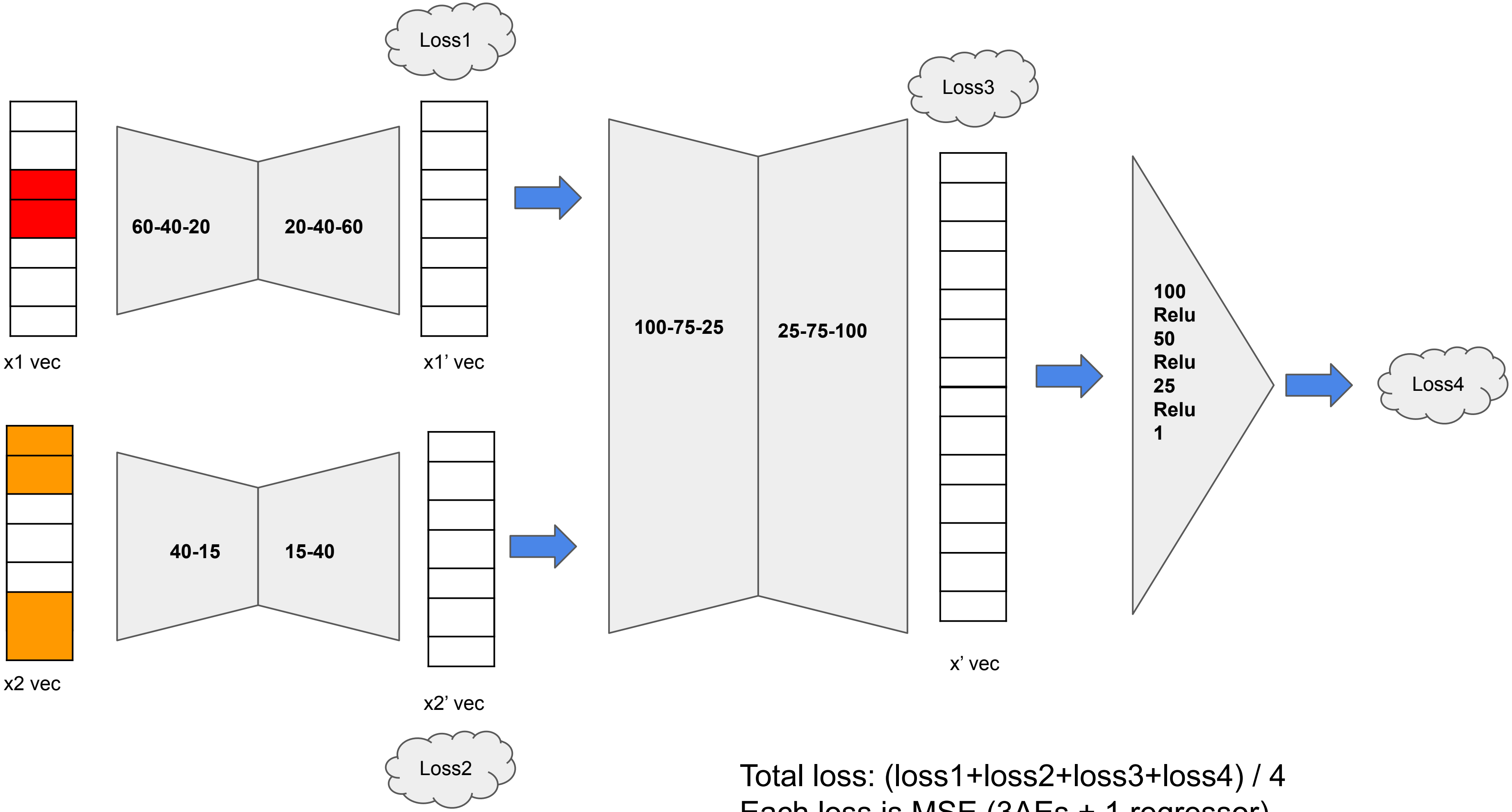
- Design an NN based solution for this problem
- A straightforward solution is to concatenate x1+x2 and do regression
  - We can simulate the missing data in our runs
  - Cool but we found that the network doesn't work well
    - Our analysis: this might be due to have many missing data during inference
- Behind the scenes
  - I want to motivate **architecture design skills** in deep learning
  - So think in building a complex network structure

Data Simulation

- We can’t just train the network with the data as it is
- We simply simulate missing data up to 90 %of the features
  - In each batch, pick a random number in range r = [0-0.9]
  - Say r = 0.5
  - Drop 50% of the features for this example

Design

- Seems our core challenge is the missing data
  - The basic model couldn’t learn how to handle such excessive missing data
- What about recovering them?
- What if we have a denoising autoencoder that takes input feature of missing features and recover them?
- In fact, we can have 3 autoencoders?
  - One for source X1, another for X2 and their for their concatenation
  - We have 3 losses from here
- Then the finally recovered vector is feed to a regressor!
  - We have 1 loss from here



Total loss: (loss1+loss2+loss3+loss4) / 4
Each loss is MSE (3AEs + 1 regressor)

```python
class AutoencoderX1(nn.Module):
    def __init__(self):
        super(AutoencoderX1, self).__init__()
        self.encoder = nn.Sequential(
            nn.Linear(in_features=60, out_features=40),
            nn.Linear(in_features=40, out_features=20),
        )
        self.decoder = nn.Sequential(
            nn.Linear(in_features=20, out_features=40),
            nn.Linear(in_features=40, out_features=60),
        )

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

```python
class AutoencoderX2(nn.Module):
    def __init__(self):
        super(AutoencoderX2, self).__init__()
        self.encoder = nn.Sequential(
            nn.Linear(in_features=40, out_features=15),
        )
        self.decoder = nn.Sequential(
            nn.Linear(in_features=15, out_features=40),
        )

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

```python
class AutoencoderX(nn.Module):
    def __init__(self):
        super(AutoencoderX, self).__init__()
        self.encoder = nn.Sequential(
            nn.Linear(in_features=100, out_features=75),
            nn.Linear(in_features=75, out_features=25),
        )
        self.decoder = nn.Sequential(
            nn.Linear(in_features=25, out_features=75),
            nn.Linear(in_features=75, out_features=100),
        )

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

- We can merge all these components together
- Return all intermediate results for the different losses
- If you want to do a skip connection, we simple add what we want it skipped

```
class AutoencoderMerger(nn.Module):
    def __init__(self):
        super(AutoencoderMerger, self).__init__()
        self.ae_x1 = AutoencoderX1()
        self.ae_x2 = AutoencoderX2()
        self.ae_x = AutoencoderX()

    def forward(self, x1, x2):
        x1 = self.ae_x1(x1)
        x2 = self.ae_x2(x2)
        x = torch.hstack([x1, x2])

        # x+ for skip connection (just element wise addition)
        x = x + self.ae_x(x)

        return x1, x2, x
```

```python
class RegressionModel(nn.Module):
    def __init__(self):
        super(RegressionModel, self).__init__()
        self.autoencoder = AutoencoderMerger()
        self.network = nn.Sequential(
            nn.Linear(100, 50),
            nn.ReLU(),
            nn.Linear(50, 25),
            nn.ReLU(),
            nn.Linear(25, 1)
        )

    def forward(self, x1, x2):
        x1, x2, x = self.autoencoder(x1, x2)
        prediction = self.network(x)
        return x1, x2, x, prediction     # apply losses
```

```python
def drop_features(x, drop_prob):
    drop_prob = torch.rand(1) * drop_prob
    # Create a mask using a uniform distribution and the drop probability
    mask = torch.rand_like(x) > drop_prob
    dropped_x = x * mask.float()
    return dropped_x
```

```python
drop_prob = 0.9
criterion = nn.MSELoss()
regressor = RegressionModel()

x1 = torch.randn(32, 60)  # Batch size of 32
x2 = torch.randn(32, 40)  # Batch size of 32
x = torch.hstack([x1, x2])
gt = torch.randn(32, 1)

x1_noise = drop_features(x1, drop_prob)
x2_noise = drop_features(x2, drop_prob)

x1_rec, x2_rec, x_rec, prediction = regressor(x1_noise, x2_noise)

# observe loss with x1 NOT x1_noise
loss1 = criterion(x1, x1_rec)
loss2 = criterion(x2, x2_rec)
loss3 = criterion(x, x_rec)
loss4 = criterion(gt, prediction)
loss = (loss1+loss2+loss3+loss4)/4.0
...
```

Welcome to Deep Learning

- We design complex networks where we may have different parts of the network designed for specific goals
- Multiple losses help us improve all our aspect

"Acquire knowledge and impart it to the people."

"Seek knowledge from the Cradle to the Grave."