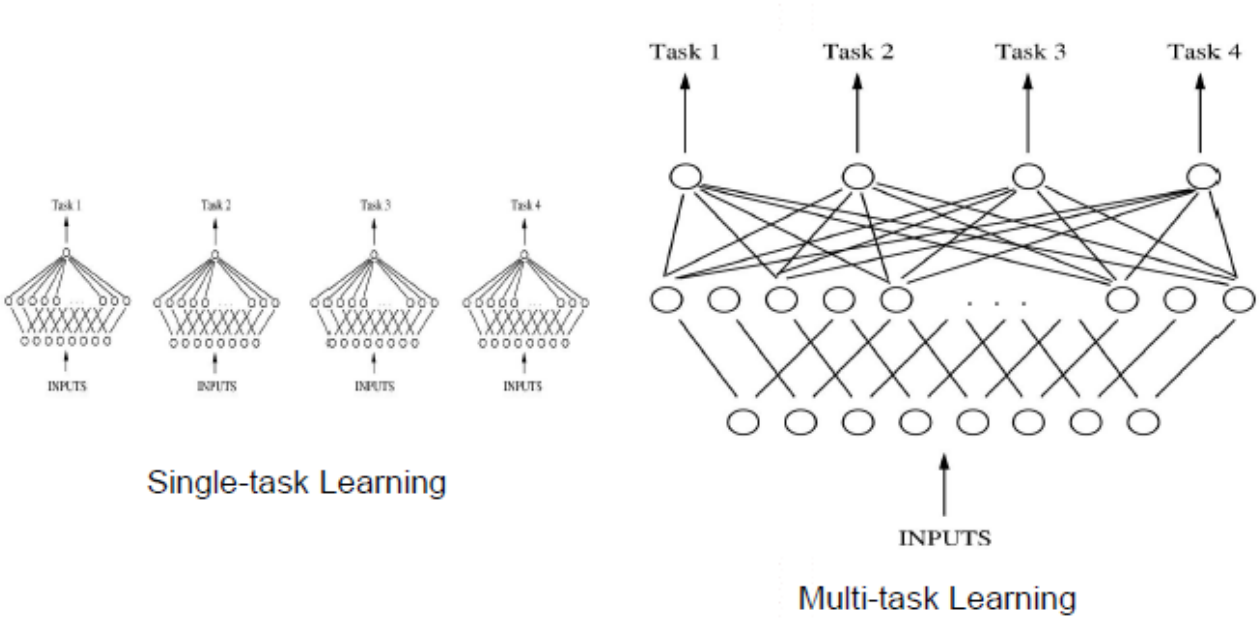
Lec11_RNN

Multi-task learning(MTL, ML tech)

- **improve generalization performance** of learning task by jointly learning multiple-related task
- success MTL→ task need to be related, share part of representation
- help transfer knowledge among task since it leverage training data more efficiently



N shot learning (Few shot, Zero shot)

- **Few shot Learning** classify classes with only a few examples
- **One shot Learning** classify classes with only one example
- **Zero shot Learning**: classify unseen classes without any examples
- **N way K Shot classification** problem classify N classes each with K examples

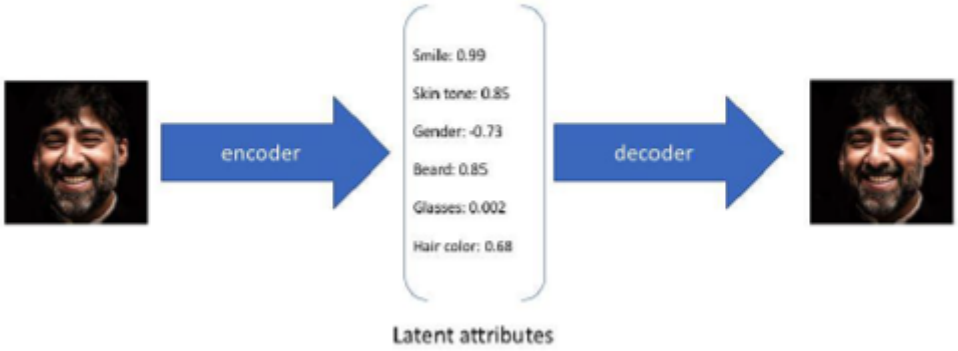
Generative networks

Variational Auto Encoder(VAE)

- **training**: input data compressed into a vector of low dimensional normal distribution by the encder, the vector is reconstructed into the input data by the decoder.

- **generation:** a vector of normal distribution is randomly sampled and input into the decoder to

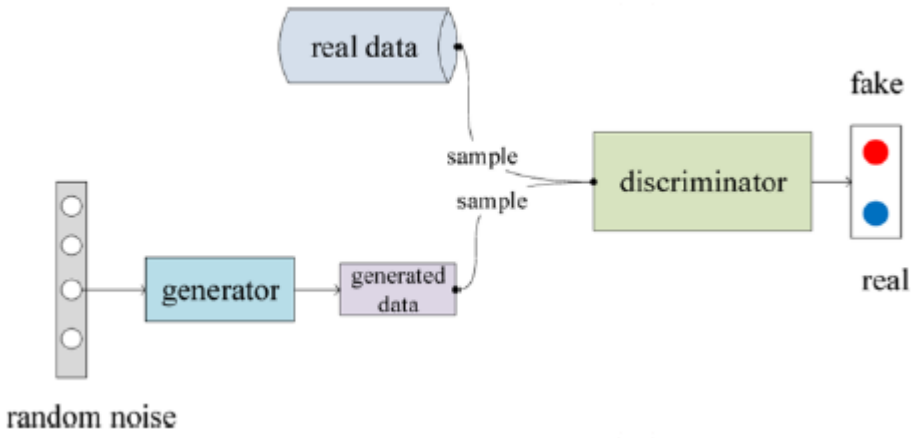
$$\text{loss} = \|x - \hat{x}\|^2 = \|x - d(z)\|^2 = \|x - d(e(x))\|^2$$



generate data

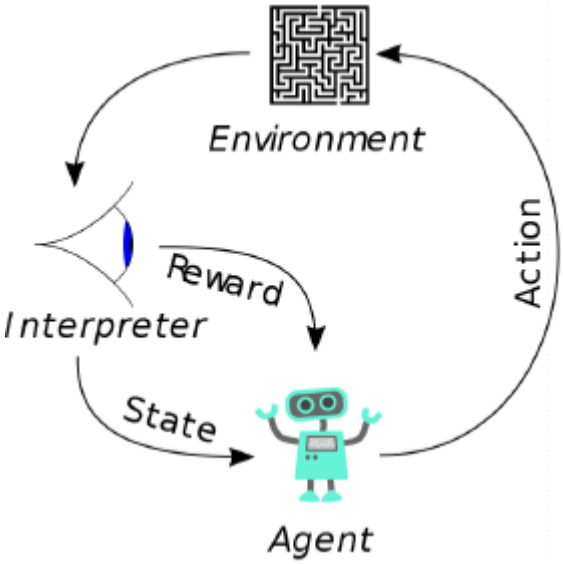
- output of VAE encoder are 2 value: mean & var(X). Modify vector E that follows standard N(0,1) by adding the mean values & multiplying by the variance. The modified vector is used as input of decoder for generation.(reparameterization重新参数化) → avoid situation that sampling is non-differentiable → no gradient

GAN(Generative Adversarial Networks



- unsupervised generation model,
- generator generate fake data to fool the discriminator
- & discriminator鉴别avoid being fooled (opponent)
- application: style transfer, img generation

Reinforcement learning



Tutorial

```
# Generate samples
np.random.seed(2)

T = 20
L = 1000
N = 100

x = np.empty((N, L), 'int64')
x[:] = np.array(range(L)) + np.random.randint(-4 * T, 4 * T, N).reshape(N, 1)
data = np.sin(x / 1.0 / T).astype('float64')
torch.save(data, open('traindata.pt', 'wb'))

# define a RNN model
class Sequence(nn.Module):
    def __init__(self):
        super(Sequence, self).__init__()
        self.lstm1 = nn.LSTMCell(1, 51)
        self.lstm2 = nn.LSTMCell(51, 51)
        self.linear = nn.Linear(51, 1)

    def forward(self, input, future = 0):
        outputs = []
        h_t = torch.zeros(input.size(0), 51, dtype=torch.double)
        c_t = torch.zeros(input.size(0), 51, dtype=torch.double)
        h_t2 = torch.zeros(input.size(0), 51, dtype=torch.double)
        c_t2 = torch.zeros(input.size(0), 51, dtype=torch.double)

        for input_t in input.split(1, dim=1):
            h_t, c_t = self.lstm1(input_t, (h_t, c_t))
            h_t2, c_t2 = self.lstm2(h_t, (h_t2, c_t2))
            output = self.linear(h_t2)
            outputs += [output]
        for i in range(future):# if we should predict the future
            h_t, c_t = self.lstm1(output, (h_t, c_t))
```

```
            h_t2, c_t2 = self.lstm2(h_t, (h_t2, c_t2))
            output = self.linear(h_t2)
            outputs += [output]
        outputs = torch.cat(outputs, dim=1)
        return outputs

steps = 15
# set random seed to 0
np.random.seed(0)
torch.manual_seed(0)
# load data and make training set
data = torch.load('traindata.pt')
input = torch.from_numpy(data[3:, :-1])
target = torch.from_numpy(data[3:, 1:])
test_input = torch.from_numpy(data[:3, :-1])
test_target = torch.from_numpy(data[:3, 1:])
# build the model
seq = Sequence()
seq.double()
criterion = nn.MSELoss()
# use LBFGS as optimizer since we can load the whole data to train
optimizer = optim.LBFGS(seq.parameters(), lr=0.4)
#begin to train
for i in range(steps):
    print('STEP: ', i)
    def closure():
        optimizer.zero_grad()
        out = seq(input)
        loss = criterion(out, target)
        print('loss:', loss.item())
        loss.backward()
        return loss
    optimizer.step(closure)
    # begin to predict, no need to track gradient here
    with torch.no_grad():
        future = 1000
        pred = seq(test_input, future=future)
        loss = criterion(pred[:, :-future], test_target)
        print('test loss:', loss.item())
        y = pred.detach().numpy()
    # draw the result
    plt.figure(figsize=(30,10))
    plt.title('Predict future values for time sequences\n(Dashlines are predicted
values)', fontsize=30)
    plt.xlabel('x', fontsize=20)
    plt.ylabel('y', fontsize=20)
    plt.xticks(fontsize=20)
    plt.yticks(fontsize=20)
    def draw(yi, color):
        plt.plot(np.arange(input.size(1)), yi[:input.size(1)], color, linewidth =
2.0)
        plt.plot(np.arange(input.size(1), input.size(1) + future),
yi[input.size(1):], color + ':', linewidth = 2.0)
    draw(y[0], 'r')
```

```
        draw(y[1], 'g')
        draw(y[2], 'b')
        plt.show()
        plt.savefig('predict%d.png'%i)
        plt.close()
```