SWWAE

Baris Aksakal

November 2023

1 Paper

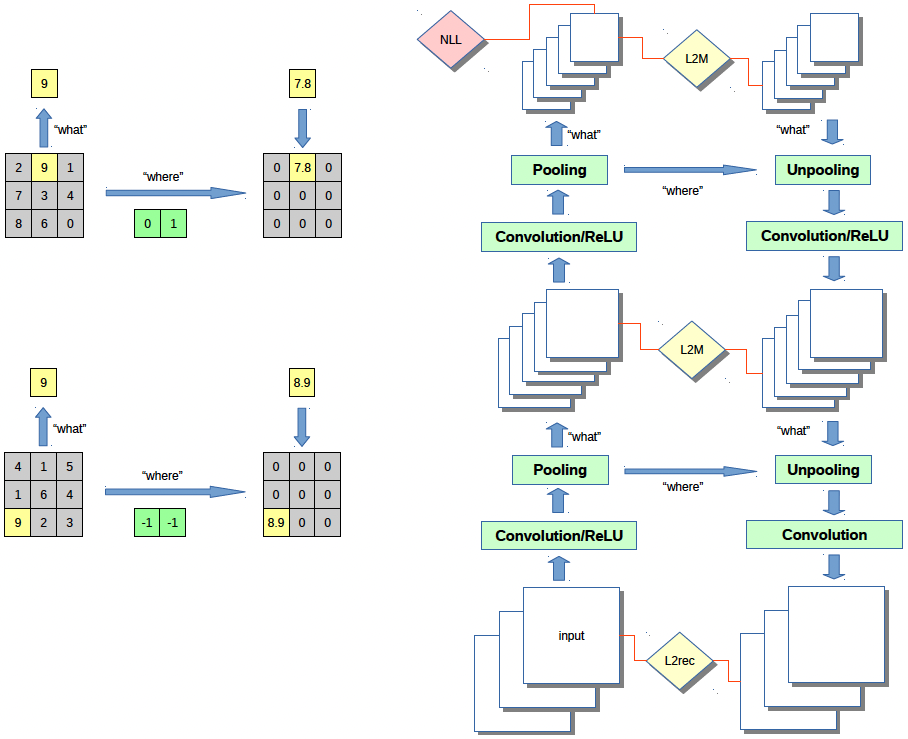
1.1 Main Idea

Supervised, semi-supervised, and unsupervised learning without relying on sampling during training.

1.2

Sampling During Training: Often used in probabilistic models and algorithms like Markov Chain Monte Carlo (MCMC) methods, where the model samples from the probability distribution of the data to make inferences or predictions. However, this can introduce noise into the gradient estimates during backpropagation, making it challenging to train large scale networks efficiently. The SWWAE architecture avoids this by not requiring sampling during the training process, instead using a deterministic approach for both the feed-forward and feedback (generative) paths.

1.3



Our model attempts to satisfy two objectives: (i)-to learn a factorized representation that encodes invariance and equivariance, (ii)-we want to leverage both labeled and unlabeled data to learn this representation in a unified framework.

1.3.1 Factorized Representation Encoding Invariance and Equivariance:

Invariance (WHAT?): Representation does not change when variations irrelevant to the task at hand are applied to the input. For instance, a face recognition system should recognize a face regardless of the lighting or the angle from which the face is viewed.

For MNIST, invariance would mean that the model should recognize the digit '2', for example, regardless of slight variations in how it's written by different people. Some people might write '2' with a loop, others without; some with a slant, others more upright. An invariant model would abstract away these differences and focus on the core features that make '2' identifiable as '2', irrespective of individual handwriting quirks, the size of the digit, or its orientation

on the image canvas.

Equivariance (WHERE?): The property of a system where transformations applied to the input change the output in a predictable way. For example, if you shift an image of a cat to the right, an equivariant system would also shift the representation of the cat to the right.

For MNIST, Equivariance would relate to how a shift in the digit's position within the image would result in a proportional shift in the model's internal representation of that image. If an image of the digit '2' is shifted to the right by a few pixels, an equivariant model would also shift its internal representation of the '2' to the right. This is important for convolutional neural networks (CNNs), which are often used with MNIST, because the convolution operation is equivariant to translation. If the digit appears in a different part of the image, the convolutional filters will activate correspondingly in a different part of the feature map.

1.3.2 Leveraging Both Labeled and Unlabeled Data:

The ability of the model to learn from both labeled data (where the desired output is known) and unlabeled data (where the desired output is not known). This is crucial for semi-supervised learning where there's a small amount of labeled data and a large amount of unlabeled data. A unified framework means that the same model architecture and learning procedure can be applied regardless of whether the data is labeled or not.

1.4 Training

The main idea of the approach we propose here is very simple: whenever a layer implements a many-to-one mapping, we compute a set of complementary variables that enable reconstruction. A schematic of our model is depicted in figure 1 (b) [-of the paper]. In the max-pooling layers of Convnets, we view the position of the max-pooling "switches" as the complementary information necessary for reconstruction. The reconstruction penalty at each layer constrains the hidden states of the feed-back pathway to be close to the hidden states of the feed-forward pathway

1.4.1 Training Modes

The system can be trained in purely supervised manner: the bottom input of the feed-forward pathway is given the input, the top layer of the feed-back pathway is given the desired output, and the weights of the decoders are updated to minimize the sum of the reconstruction costs. If only the top-level cost is used, the model reverts to purely supervised backprop. If the hidden layer reconstruction costs are used, the model can be seen as supervised with a reconstruction

regularization. In unsupervised mode, the top-layer label output is left un- constrained, and simply copied from the output of the feed-forward pathway. The model becomes a stacked convolutional auto-encoder.

1.4.2 Loss Function

$$L = L_{NLL} + \lambda_{L2rec} L_{L2rec} + \lambda_{L2M} L_{L2M}$$

Pooling layers in the encoder split information into "what" and "where" components, depicted in fig- ure 1(a), that "what" is essentially max and "where" carries argmax, i.e., the switches of maximally activation defined under local coordinate frame over each pooling region. The "what" component is fed upward through the encoder, while the "where" is fed through lateral connections to the same stage in the feed-back decoding pathway. The decoder uses convolution and "unpooling" opera- tions to approximately invert the output of the encoder and reproduce the input, shown in figure 1. The unpooling layers use the "where" variables to unpool the feature maps by placing the "what" into the positions indicated the preserved switches. We use negative log-likelihood (NLL) loss for classification and L2 loss for reconstructions; e.g, L = LNLL +

$$L_{L2rec} = \|x - \tilde{x}\|_2, \qquad L_{L2M} = \|x_m - \tilde{x}_m\|_2,$$

L2recLL2rec + L2M LL2M , (1) where LNLL is the discriminative loss, LL2rec is the reconstruction loss at the input level and LL2M charges intermediate reconstruction terms. (lamda)'s weight the losses against each other.

1.4.3 SOFT VERSION "WHAT" AND "WHERE"

Recently, Goroshin et al. (2015) introduces a soft version of max and argmax operators within each pooling region. The soft pooling and unpooling can be embedded seamlessly into the SWWAE model and it has the virtue such that it can backpropogate through p, in the contrast to the hard max-pooling being not differentiable w.r.t the argmax "switch" locations. Furthermore, soft-pooling operators enable location information to be more accurately represented and thus enable the features to capture fine details about the input, as evidenced in our visualization experiments (see section 4.2).

1.4.4 INTERMEDIATE L2 CONSTRAINTS

The reasons for adding intermediate L2 reconstruction terms are listed as follow. First, it prevents the feature planes from being shuffled so that the "where" map

$$m_k = \sum_{N_k} z(x,y) \frac{e^{\beta z(x,y)}}{\sum_{N_k} e^{\beta z(x,y)}} \approx \max_{N_k} z(x,y)$$

$$\mathbf{p_k} = \sum_{N_k} \begin{bmatrix} x \\ y \end{bmatrix} \frac{e^{\beta z(x,y)}}{\sum_{N_k} e^{\beta z(x,y)}} \approx \arg\max_{N_k} z(x,y),$$

conveyed from encoder ith are guaranteed to match the "what" from decoder ith. Otherwise, the unpooling may see "what" and "where" with shuffle orders, and hence cannot work properly. Second, in particular when training with classification loss, intermediate terms disallow the scenario that upper layers become idle while only lower layers are busy at reconstructing, in which case filters from those unemployed layers are not regularized.

1.5 What-Where Benefits

"What" learns the content or features in the data. In a visual task, for instance, this would be the presence of objects, shapes, or textures in an image. It's the part of the data representation that captures the essence of the data itself.

"Where" learns the spatial or temporal information about the features. This includes the location of objects in an image or the sequence of events in a time series. It's the part of the data representation that captures the positional or timing context of the features identified by the "what".

3.2 Training with joint losses and regularization

As we mentioned, the SWWAE provides a unified framework for learning with all three learning modalities, all within a single architecture and single learning algorithm, i.e. stochastic gradient descent and backprop. Switching between these modalities can be achieved as follows:

- for supervised learning, we can mask out the entire Deconvnet pathway by setting λ_{L2*} to 0 and the SWWAE falls back to vanilla Convnet.
- for unsupervised learning, we can nullify the fully-connected layers on top of Convnet together with softmax classifier by setting $\lambda_{NLL} = 0$. In this setting, the SWWAE is equivalent to a deep convolutional auto-encoder.
- for semi-supervised learning, all three terms of the loss are active. The gradient contributions from the Deconvnet can be interpreted as an information preserving regularizer.

1.6 More on Training Modes wrt Loss Formulation

2 Code

```
class MaxPoolingWithArgmax2D(Layer):
    def __init__(
            self,
            pool_size=(2, 2),
            strides=(2, 2),
            padding='same',
            **kwargs):
        super(MaxPoolingWithArgmax2D, self).__init__(**kwargs)
        self.padding = padding
        self.pool_size = pool_size
        self.strides = strides

    def call(self, inputs, **kwargs):
        padding = self.padding
        pool_size = self.pool_size
        strides = self.strides
        ksize = [1, pool_size[0], pool_size[1], 1]
        strides = [1, strides[0], strides[1], 1]
        output, argmax = tf.nn.max_pool_with_argmax(inputs, ksize, strides, 'SAME')
        argmax = tf.cast(argmax, tf.keras.backend.floatx())
        return [output, argmax]

    def compute_output_shape(self, input_shape):
        ratio = (1, 2, 2, 1)
        output_shape = [
            dim // ratio[idx]
            if dim is not None else None
            for idx, dim in enumerate(input_shape)]
        output_shape = tuple(output_shape)
        return [output_shape, output_shape]

    def compute_mask(self, inputs, mask=None):
        return 2 * [None]
```

class MaxPoolingWithArgmax2D(Layer): A custom layer that performs max pooling on the input and also records the locations (indices) of the maxi-

mum values. This is used later to reconstruct the input data from its compressed form.

def call(self, inputs, **kwargs): This is where the actual max pooling operation takes place. It takes the input data, applies max pooling, and returns both the pooled data and the locations of the maximum values.

```
class MaxUnpooling2D(Layer):
    def __init__(self, up_size=(2, 2), **kwargs):
        super(MaxUnpooling2D, self).__init__(**kwargs)
        self.up_size = up_size

    def unpool(self, pool, ind, ksize=[1, 2, 2, 1]):
        input_shape = tf.shape(pool)
        output_shape = [input_shape[0], input_shape[1] * ksize[1], input_shape[2] * ksize[2], input_shape[3]]

        flat_input_size = tf.math.reduce_prod(input_shape)
        flat_output_shape = [output_shape[0], output_shape[1] * output_shape[2] * output_shape[3]]

        pool_ = tf.reshape(pool, [flat_input_size])
        batch_range = tf.reshape(tf.range(tf.cast(output_shape[0], tf.int64), dtype=ind.dtype),
                                 shape=[input_shape[0], 1, 1, 1])
        b = tf.ones_like(ind) * batch_range
        b1 = tf.reshape(b, [flat_input_size, 1])
        ind_ = tf.reshape(ind, [flat_input_size, 1])
        ind_ = tf.concat([b1, ind_], 1)

        ret = tf.scatter_nd(ind_, pool_, shape=tf.cast(flat_output_shape, tf.int64))
        ret = tf.reshape(ret, output_shape)

        set_input_shape = pool.get_shape()
        set_output_shape = [set_input_shape[0], set_input_shape[1] * ksize[1], set_input_shape[2] * ksize[2],
        ret.set_shape(set_output_shape)
        return ret

    def call(self, inputs, output_shape=None):
        updates = inputs[0]
        mask    = tf.cast(inputs[1], dtype=tf.int64)
        ksize   = [1, self.up_size[0], self.up_size[1], 1]
        return self.unpool(updates, mask, ksize)

    def compute_output_shape(self, input_shape):
        mask_shape = input_shape[1]
        return (
            mask_shape[0],
            mask_shape[1] * self.up_size[0],
            mask_shape[2] * self.up_size[1],
            mask_shape[3]
        )
```

class MaxUnpooling2D(Layer): The counterpart to MaxPoolingWithArgmax2D, this custom layer takes the compressed data and the recorded indices to place the values back into their original locations, effectively "unpooling" the data.

def call(self, inputs, output_shape=None): Performs the unpooling operation using the pooled data and recorded indices to reconstruct the original size of the data.

```
class Encoder(Model):
    def __init__(self, input_dim):
        super(Encoder, self).__init__()
        self.input_1 = Input(shape=input_dim)
        self.conv1 = Conv2D(16, (5, 5), activation='relu', padding='same')
        self.conv2 = Conv2D(32, (3, 3), activation='relu', padding='same')
        self.max_pool_argmax = MaxPoolingWithArgmax2D(pool_size=(7,7), strides=(7

    def call(self, x):
        #x = self.input_1(x)
        x = self.conv1(x)
        x = self.conv2(x)
        x, mask_1 = self.max_pool_argmax(x)
        L2M_encoder_1 = x
        return x, mask_1, L2M_encoder_1
```

```
class Decoder(Model):
    def __init__(self, original_shape):
        super(Decoder, self).__init__()
        #self.input = Input(shape=input_dim)
        self.unpool_1 = MaxUnpooling2D((7,7))
        self.conv_1 = Conv2D(16, (3, 3), activation='relu', padding='same')
        self.conv_2 = Conv2D(1, (5, 5), activation='sigmoid', padding='same')

    def call(self, x):
        L2M_decoder_1 = x[0]
        x = self.unpool_1(x)
        x = self.conv_1(x)
        x = self.conv_2(x)
        return x, L2M_decoder_1
```

```
class SWWAE(Model):
    def __init__(self, input_dim):
        super(SWWAE, self).__init__()
        self.encoder = Encoder(input_dim)
        self.decoder = Decoder(input_dim)

    def compile(self, optimizer, loss):
        super().compile(optimizer)
        self.loss = loss

    def call(self, x):
        # Forward pass through the encoder and decoder
        encoded, mask_1, L2M_encoder_1 = self.encoder(x)
        decoded, L2M_decoder_1 = self.decoder([encoded, mask_1])
```

```
def L2rec_L2M(y_true, y_pred, L2M_encoder_1, L2M_decoder_1):
    l2m_1 = tf.math.square(tf.math.subtract(L2M_encoder_1,L2M_decoder_1))
    L2rec = tf.math.square(tf.math.subtract(y_true,y_pred))
    return tf.math.add(tf.math.reduce_mean(l2m_1),tf.math.reduce_mean(L2rec))
```

```
input_dim = (28,28,1)
latent_dim = 2
model = SWWAE(input_dim)
optimizer = tf.optimizers.Adam()
loss = L2rec_L2M
model.compile(loss=loss, optimizer=optimizer)
```

def L2rec_L2M(y_true, y_pred, L2M_encoder_1, L2M_decoder_1): Defines the loss function that combines reconstruction error with a regularization term based on the encoder and decoder outputs (Unsupervised Version).