Coding Deep

Contents

- Deep learning basics
- Frameworks
- Coding: best practices
- Building wheels
- Example: seq2seq
- Go parallel

Deep Learning Basics

Statistical Learning

Model

$$y = f(x; \mathbf{w})$$

Objective

$$L(\mathbf{w}) = \frac{1}{N} \sum_{i=1}^{N} loss\{y_i, f(x_i; \mathbf{w})\}$$

$$\arg\min_{\mathbf{w}} L(\mathbf{w})$$

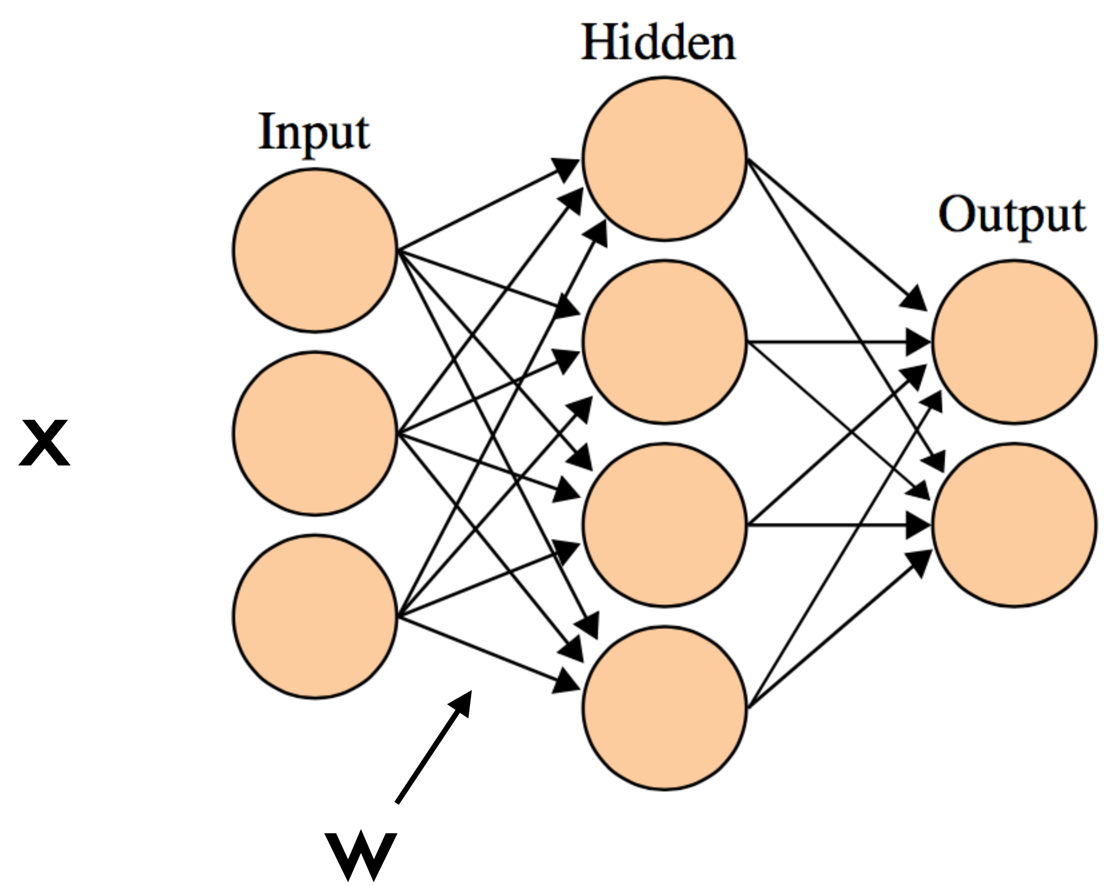
Optimization

analytical or numerical

Neural Networks

Basic cell $h(x) = \text{activation}(\mathbf{w} \cdot x + b)$

Neural Networks



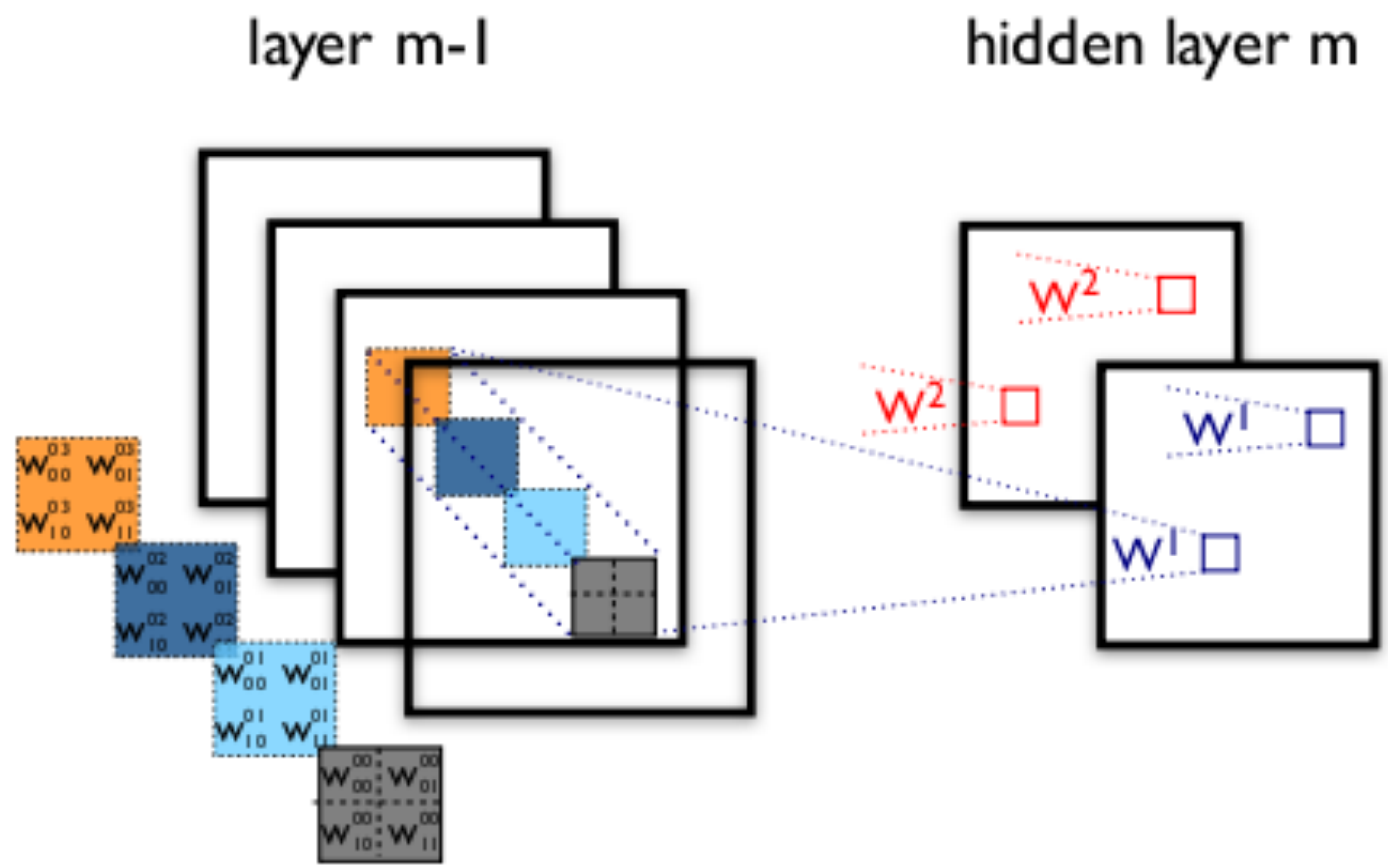
$$y = f(x; \mathbf{w})$$

MLP (multilayer perceptrons)

Neural Networks

Basic cell	$h(x) = \text{activation}(\mathbf{w} \cdot x + b)$
Convolution cell	$h(x) = \text{activation}(\mathbf{w} * x + b)$

Neural Networks

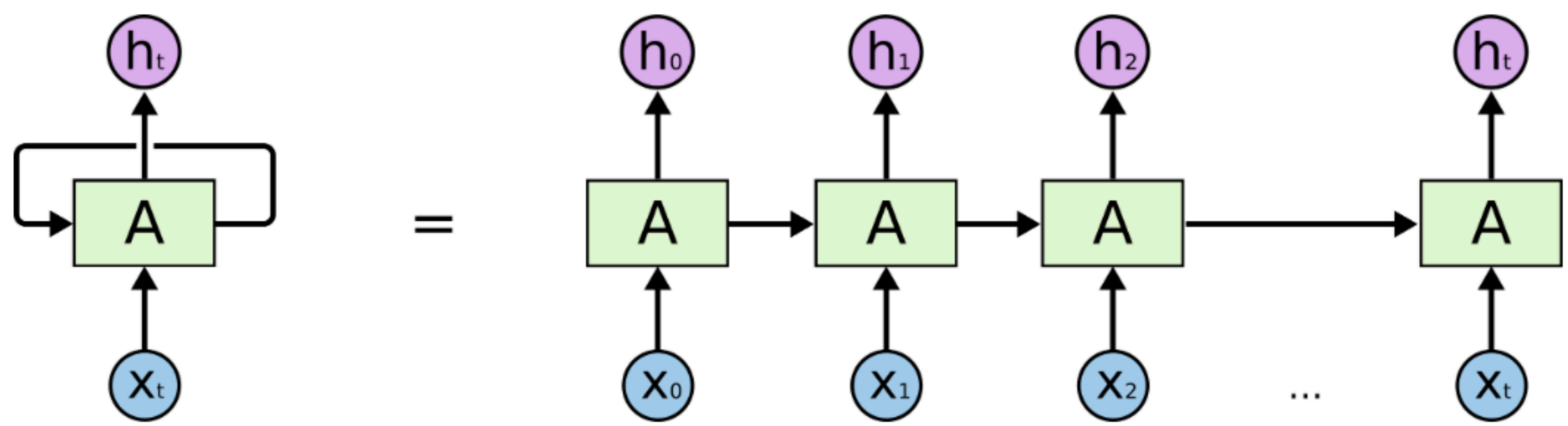


CNN (convolutional neural network)

Neural Networks

Basic cell	$h(x) = \text{activation}(\mathbf{w} \cdot x + b)$
Convolution cell	$h(x) = \text{activation}(\mathbf{w} * x + b)$
Recurrent cell	$y = \text{activation}(\mathbf{w}_{yh} \cdot h)$
	$h = \text{activation}(\mathbf{w}_{hh} \cdot h^{(last)} + \mathbf{w}_{xh} \cdot x)$

Neural Networks



RNN (recurrent neural network)

Linear Algebra Library

- Represent data as vectors/matrices/arrays
- Do linear algebra calculation

```
y_true = np.array(...)
y_pred = np.array(...)

tp = np.sum(y_pred & y_true)
precision = tp / np.sum(y_pred)
recall = tp / np.sum(y_true)
```

Algebra System

- Represent computation (computation graph)
- Calculate gradients automatically
- Utilize GPU for speed

```
a = tf.placeholder(tf.float32)
x = tf.Variable(3.)
y = x ** a
sess.run(x.initializer)
print(sess.run(tf.gradients(y, x), feed_dict={a: 2}))
```

DL Frameworks

- Provide pre-defined cells, layers, optimizers, initializers, etc.

- Simplify training process

```
model = Sequential()
model.add(Embedding(max_features, output_dim=256))
model.add(LSTM(128))
model.add(Dropout(0.5))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='binary_crossentropy', metrics=['accuracy'],
              optimizer='rmsprop')
model.fit(x_train, y_train, batch_size=16, epochs=10)
score = model.evaluate(x_test, y_test, batch_size=16)
```

DL Frameworks

Modern Frameworks

- TensorFlow (by Google)
- Keras (with Theano or TensorFlow)
- MXNet (supported by Amazon)
- PyTorch (by Facebook)

API Design

- Data input: whole array / batch / iterator
- Model definition: symbols / layers / models
- Training: step / fit
- Utilities: inspection / visualization

Example: MNIST

Inputs

- Whole array:

```
from keras.datasets import mnist
(x_train, y_train), (x_test, y_test) = mnist.load_data()
```

- Batched iterators:

```
batch_size = 100
train_iter = mx.io.NDArrayIter(train_img, train_lbl,
                               batch_size, shuffle=True)
val_iter = mx.io.NDArrayIter(val_img, val_lbl, batch_size)
```

Model Definition

- TensorFlow style
- MXNet style
- Functional style

TensorFlow Style

- Based on variables and ops

```
W_conv1 = weight_varible([5, 5, 1, 32])
b_conv1 = bias_variable([32])
h_conv1 = max_pool_2x2(tf.nn.relu(conv2d(x_image, W_conv1) + b_conv1))
```

- Model output, weight initialization, optimizer step, are all symbols

```
cross_entropy = -tf.reduce_sum(y_ * tf.log(y_conv))
train_step = tf.train.AdamOptimizer(1e-4).minimize(cross_entropy)
correct_prediction = tf.equal(tf.arg_max(y_conv, 1), tf.arg_max(y_, 1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
```

MXNet / tf.layers Style

- Also based on variables and ops
- But provides pre-defined NN layers; weights are generated automatically

```
fc1 = mx.sym.FullyConnected(data=data, name='fc1', num_hidden=128)
act1 = mx.sym.Activation(data=fc1, name='relu1', act_type="relu")
```

Functional Style

- Each layer is generated by some class, bound with specific weights
- Layers act like functions, which can be chained or stacked up

Functional Style: Keras

```
model = Sequential()
model.add(Conv2D(32, kernel_size=(3, 3),
                 activation='relu',
                 input_shape=input_shape))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(num_classes, activation='softmax'))
```

Model Reuse in Keras

```
i1 = Input(input_shape)
i2 = Input(input_shape)
o1 = model(i1)
o2 = model(i2)
# o = (o1 - o2) ** 2
o = Lambda(lambda i: K.abs(i[0] - i[1]), output_shape=output_shape)([o1, o2])
```

Functional Style: PyTorch

```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 10, kernel_size=5)
        self.conv2 = nn.Conv2d(10, 20, kernel_size=5)
        self.conv2_drop = nn.Dropout2d()
        self.fc1 = nn.Linear(320, 50)
        self.fc2 = nn.Linear(50, 10)

    def forward(self, x):
        x = F.relu(F.max_pool2d(self.conv1(x), 2))
        x = F.relu(F.max_pool2d(self.conv2_drop(self.conv2(x)), 2))
        x = x.view(-1, 320)
        x = F.relu(self.fc1(x))
        x = F.dropout(x, training=self.training)
        x = self.fc2(x)
        return F.log_softmax(x)
```

Training Process

- Step-by-step style
- Fit-on-whole-data style

Step

```python
for batch_idx, (data, target) in enumerate(train_loader):
    data, target = Variable(data), Variable(target)
    optimizer.zero_grad()
    output = model(data)
    loss = F.nll_loss(output, target)
    loss.backward()
    optimizer.step()
    if batch_idx % args.log_interval == 0:
        print('Train Epoch: {} [{}/{} ({:.0f}%)]\tLoss: {:.6f}'.format(
            epoch, batch_idx * len(data), len(train_loader.dataset),
            100. * batch_idx / len(train_loader), loss.data[0]))
```

Fit

```
model.compile(loss=keras.losses.categorical_crossentropy,
              optimizer=keras.optimizers.Adadelta(),
              metrics=['accuracy'])

model.fit(x_train, y_train,
          batch_size=batch_size,
          epochs=epochs,
          verbose=1,
          validation_data=(x_test, y_test))
```

Inspection and Evaluation

- Inspect structure
- Get weights
- Get intermediate outputs
- Save / load a model
- Logging: manually / using callbacks

Special Facilities

- Embedding
- Masking
- Normalization
- Regularization
- Label weights

More about Masking

- Masked inputs should have zero loss
- Masked terms should not be averaged

Coding a Deep Network

Coding Style is Important

We want our model to be:

- Fast
- Readable
- Reusable
- Extendible

Good coding style helps with these

Modular Design

For readability and reusability, we construct our model with these four separate parts:

- Generating inputs
- Building network
- Training
- Bookkeeping

Inputs

- Why batched?
- Use python generators (iterators)

Building a network

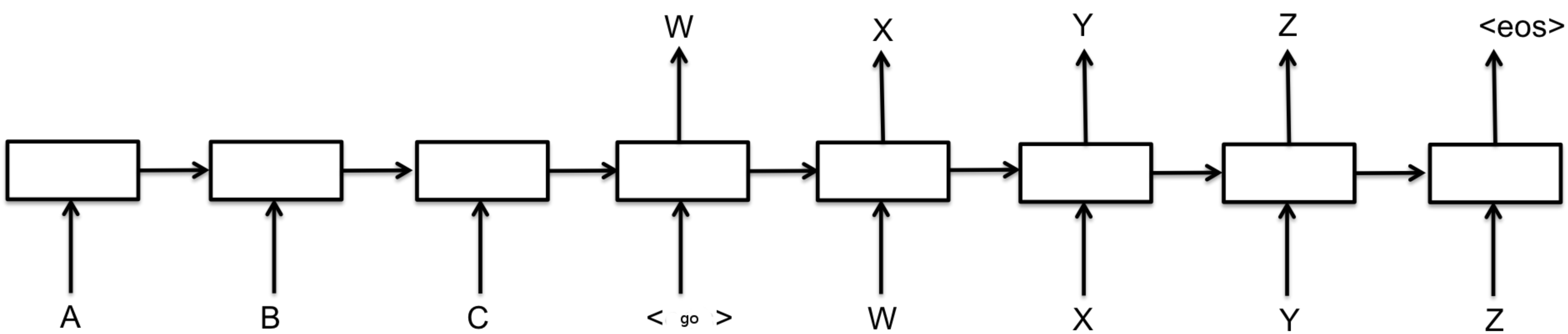
Steps:

- Defining weights / layers
- Linking up
- Shape checking
- View summary / graph

Training

- Parameter initialization
- Optimizers
- Bookkeeping: separate directory for each run

Example: seq2seq


W
X
Y
Z
<eos>
A
B
C
<go>
W
X
Y
Z

Approaches

- Word by word
- Sequence by sequence (dynamic length)
- Fixed length sequences with padding
- Bucketing

First Model: PyTorch

```python
class EncoderRNN(nn.Module):
    def __init__(self, input_size, hidden_size, n_layers=1):
        super(EncoderRNN, self).__init__()
        self.n_layers = n_layers
        self.hidden_size = hidden_size

        self.embedding = nn.Embedding(input_size, hidden_size)
        self.gru = nn.GRU(hidden_size, hidden_size)

    def forward(self, input, hidden):
        embedded = self.embedding(input).view(1, 1, -1)
        output = embedded
        for i in range(self.n_layers):
            output, hidden = self.gru(output, hidden)
        return output, hidden

    def initHidden(self):
        return Variable(torch.zeros(1, 1, self.hidden_size))
```

Attention

```python
self.attn = nn.Linear(self.hidden_size * 2, self.max_length)
self.attn_combine = nn.Linear(self.hidden_size * 2, self.hidden_size)
```

```python
attn_weights = F.softmax(self.attn(torch.cat((embedded[0], hidden[0]), 1)))
attn_applied = torch.bmm(attn_weights.unsqueeze(0),
                         encoder_outputs.unsqueeze(0))

input = torch.cat((embedded[0], attn_applied[0]), 1)
output = self.attn_combine(input).unsqueeze(0)
```

Run Model

```
for ei in range(input_length):
    encoder_output, encoder_hidden = \
        encoder(input_variable[ei], encoder_hidden)
    encoder_outputs[ei] = encoder_output[0][0]
```

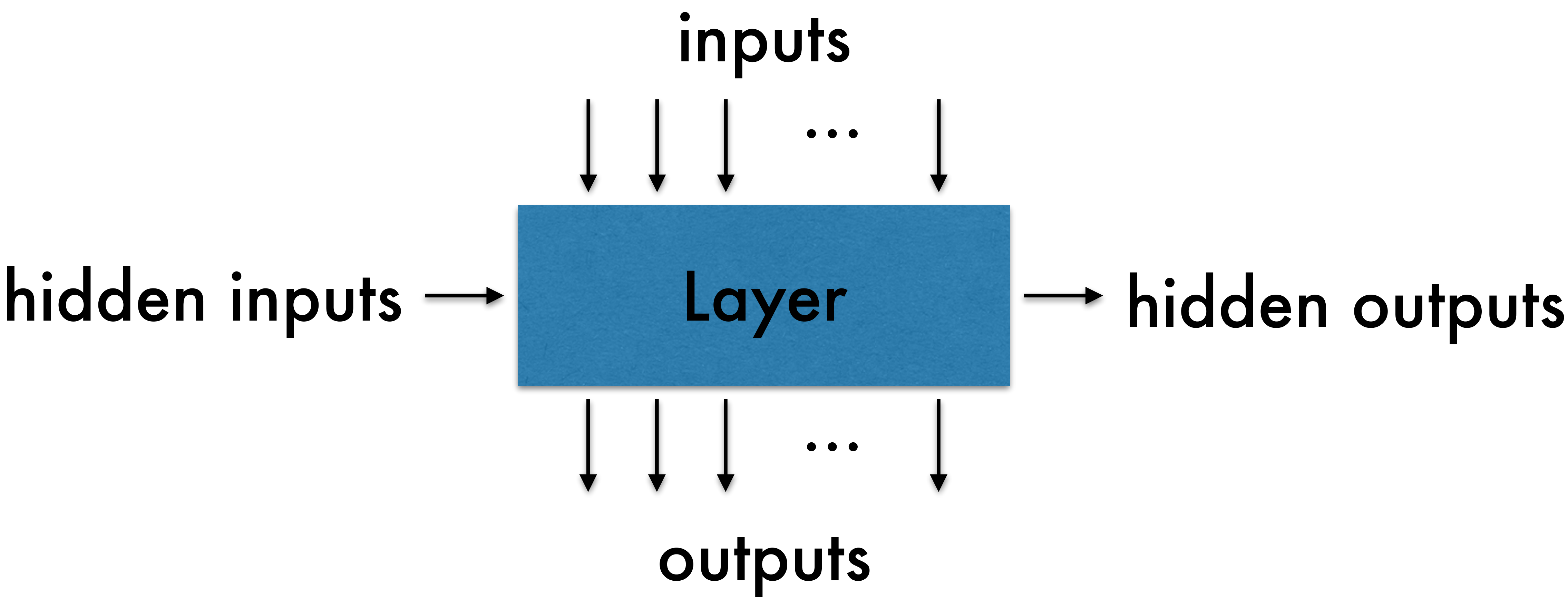
```
if use_teacher_forcing:
    # Teacher forcing: Feed the target as the next input
    for di in range(target_length):
        decoder_output, decoder_hidden, decoder_attention = \
            decoder(decoder_input, decoder_hidden,
                    encoder_output, encoder_outputs)
        loss += criterion(decoder_output[0], target_variable[di])
        decoder_input = target_variable[di]  # Teacher forcing
```

More about RNN: Stateful, Unrolling, etc.

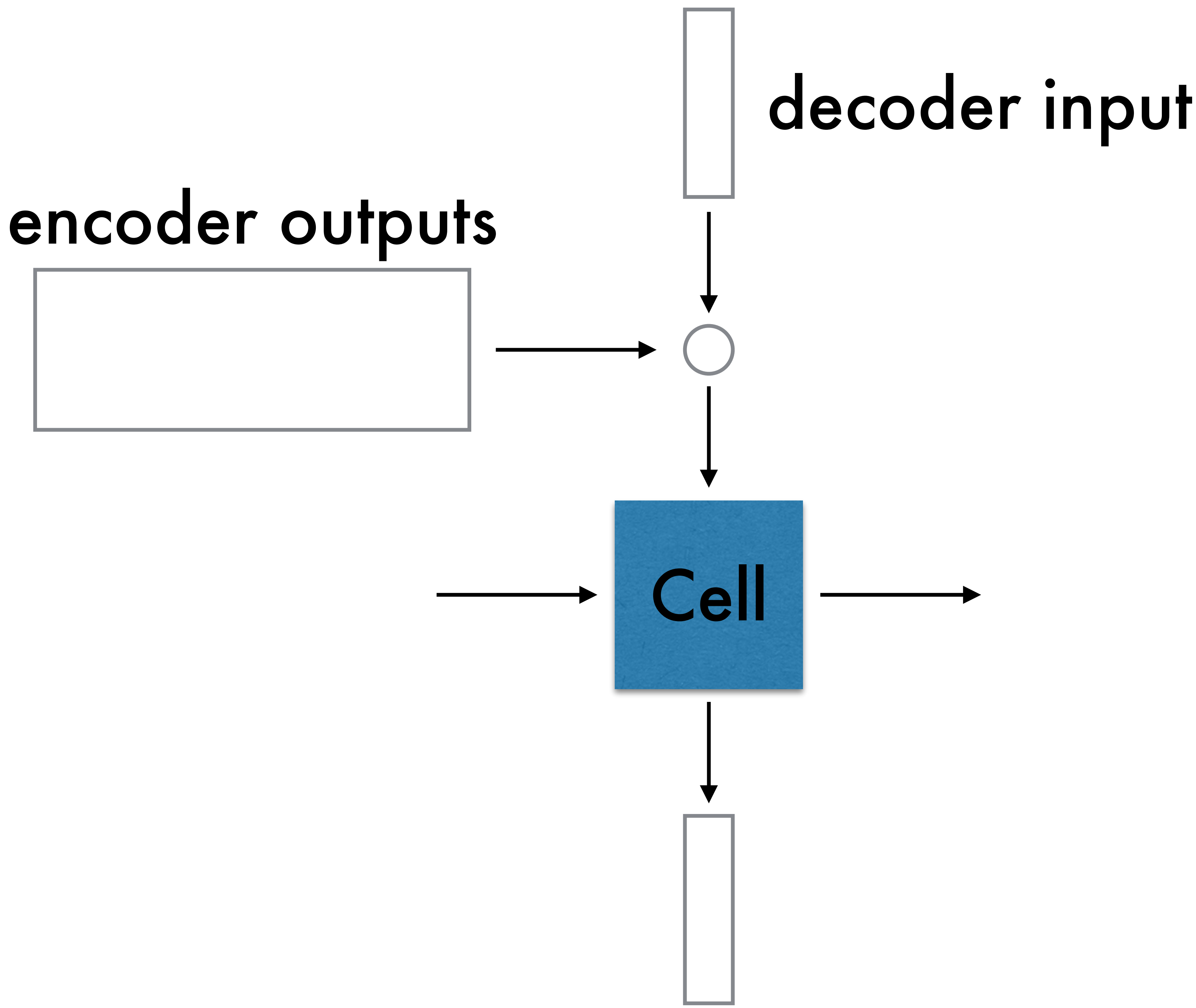
- Dynamic graph vs. static graph
- Symbolic loops vs. unrolling
- RNN cells and RNN layers
- Keras stateful API

Keras RNN Layer

- Better RNN layer from recurrentshop (on github):



Attention Decoder Cell



References

- https://github.com/fchollet/keras/issues/1579
- https://github.com/datalogai/recurrentshop
- http://mxnet.io/how_to/bucketing.html
- http://mxnet.io/architecture/note_data_loading.html
- https://github.com/farizrahman4u/seq2seq
- https://www.tensorflow.org/tutorials/seq2seq
- https://github.com/MaximumEntropy/Seq2Seq-PyTorch