Lesson 6 RNNS FROM SCRATCH

2018年7月

上海交通大學
SHANGHAI JIAO TONG UNIVERSITY

目录 Contents

上海交通大学
SHANGHAI JIAO TONG UNIVERSITY

文献分享

Entity Embeddings of Categorical Variables

Cheng Guo* and Felix Berkhahn†
Neokami Inc.
(Dated: April 25, 2016)

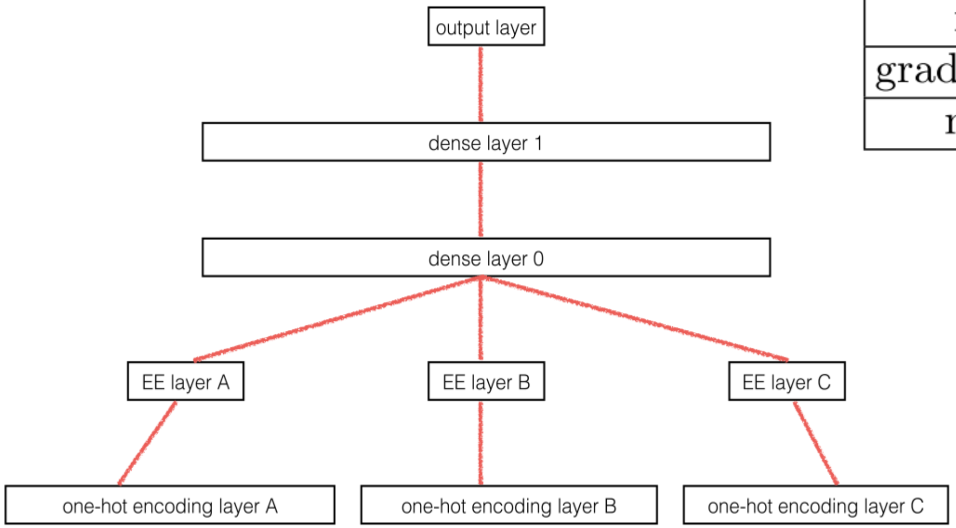
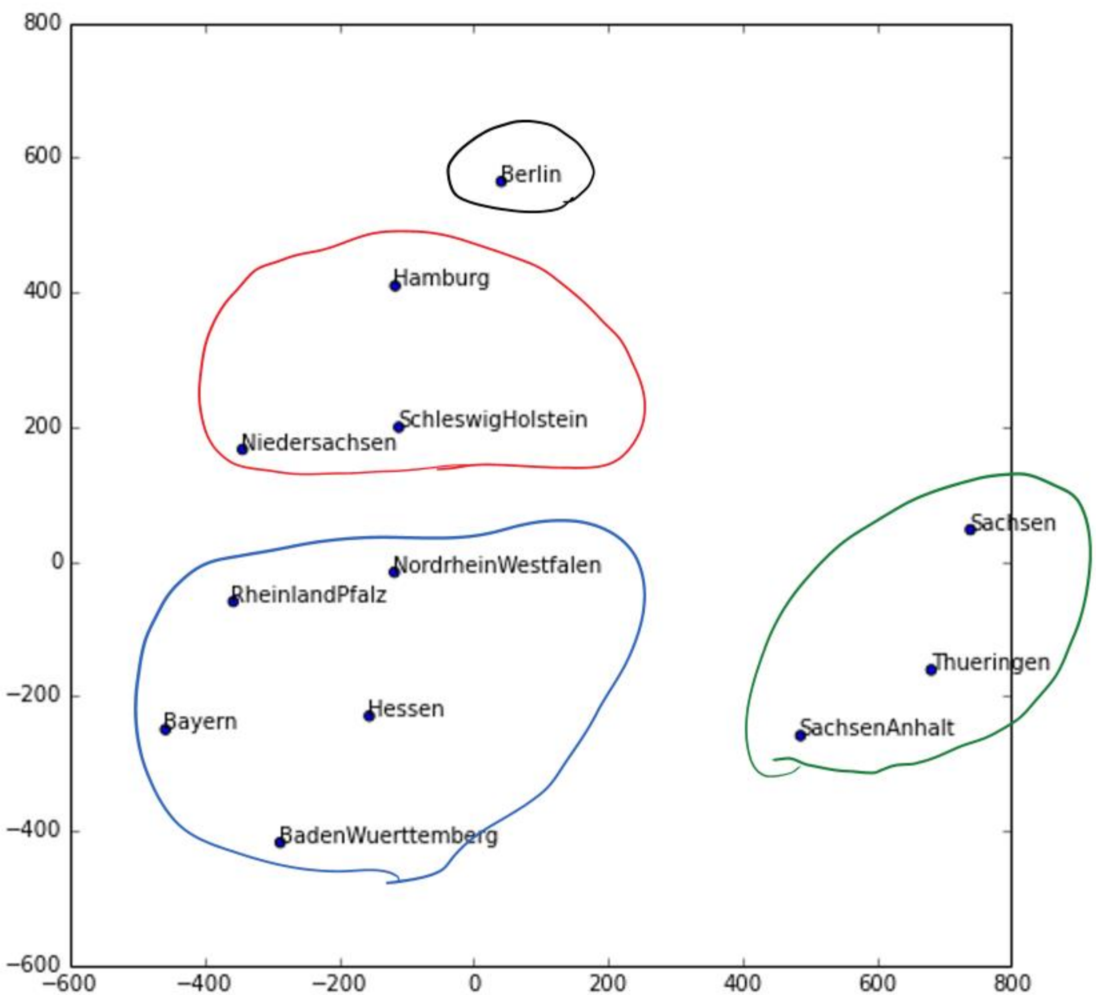


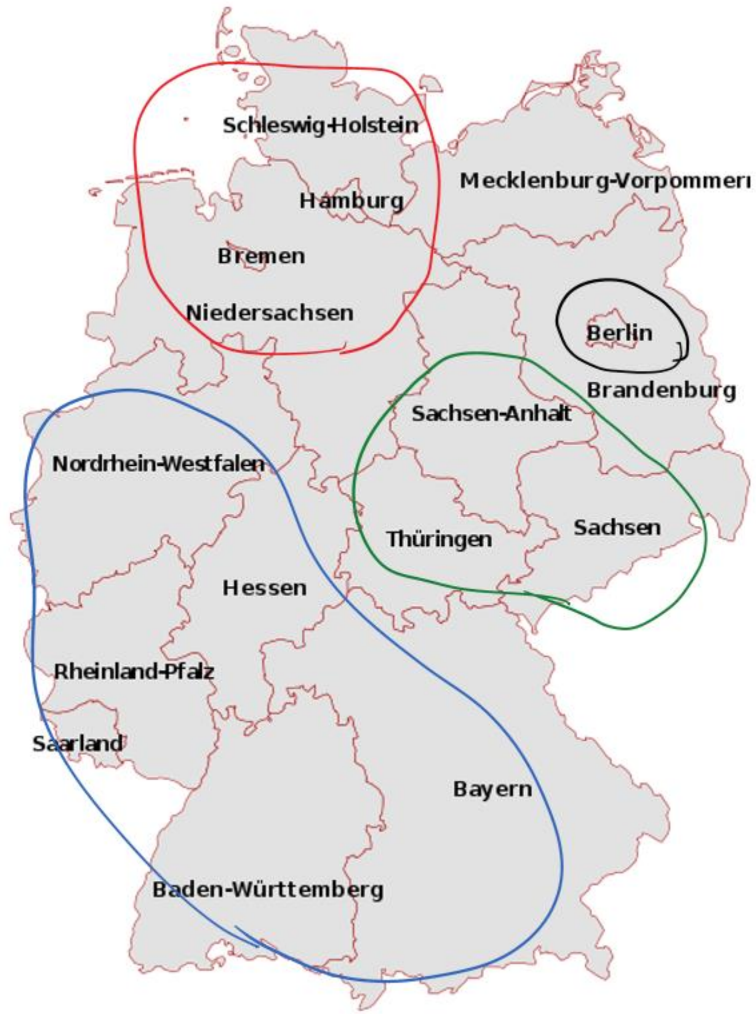
FIG. 1. Illustration that entity embedding layers are equivalent to extra layers on top of each one-hot encoded input.

method	MAPE	MAPE (with EE)
KNN	0.290	0.116
random forest	0.158	0.108
gradient boosted trees	0.152	0.115
neural network	0.101	0.093

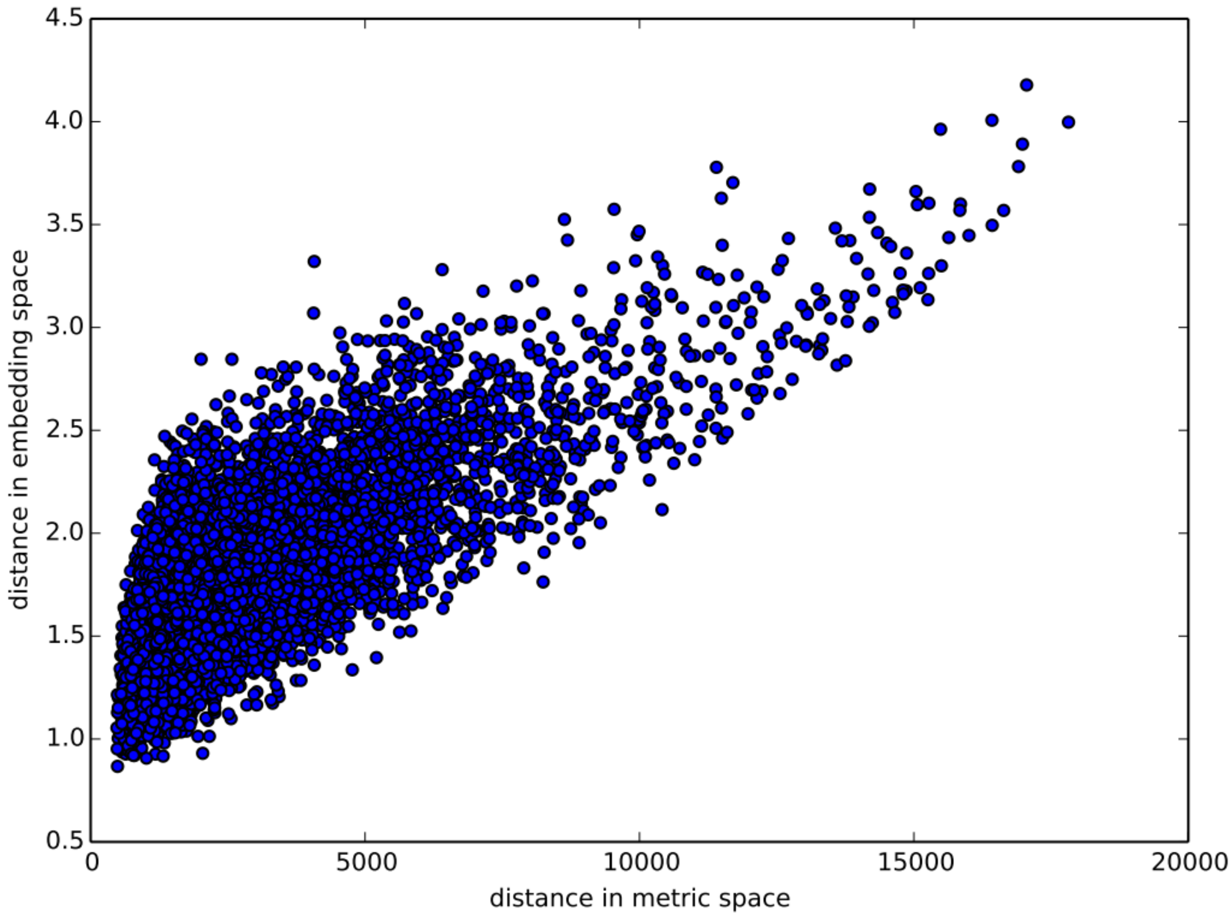
- 观察上表实验结果，我们可以通过在one-hot编码输入层之后加入实体嵌入层（EE层）的方法，来降低MAPE值（mean absolute percent error /平均绝对百分比偏差）

- 将每个嵌入向量两个最主要组成参数值（two first principal components）绘制成二维平面图。我们发现即使没有给出各个州区地理位置信息，它们在二维图上的位置和实际地理分布很接近

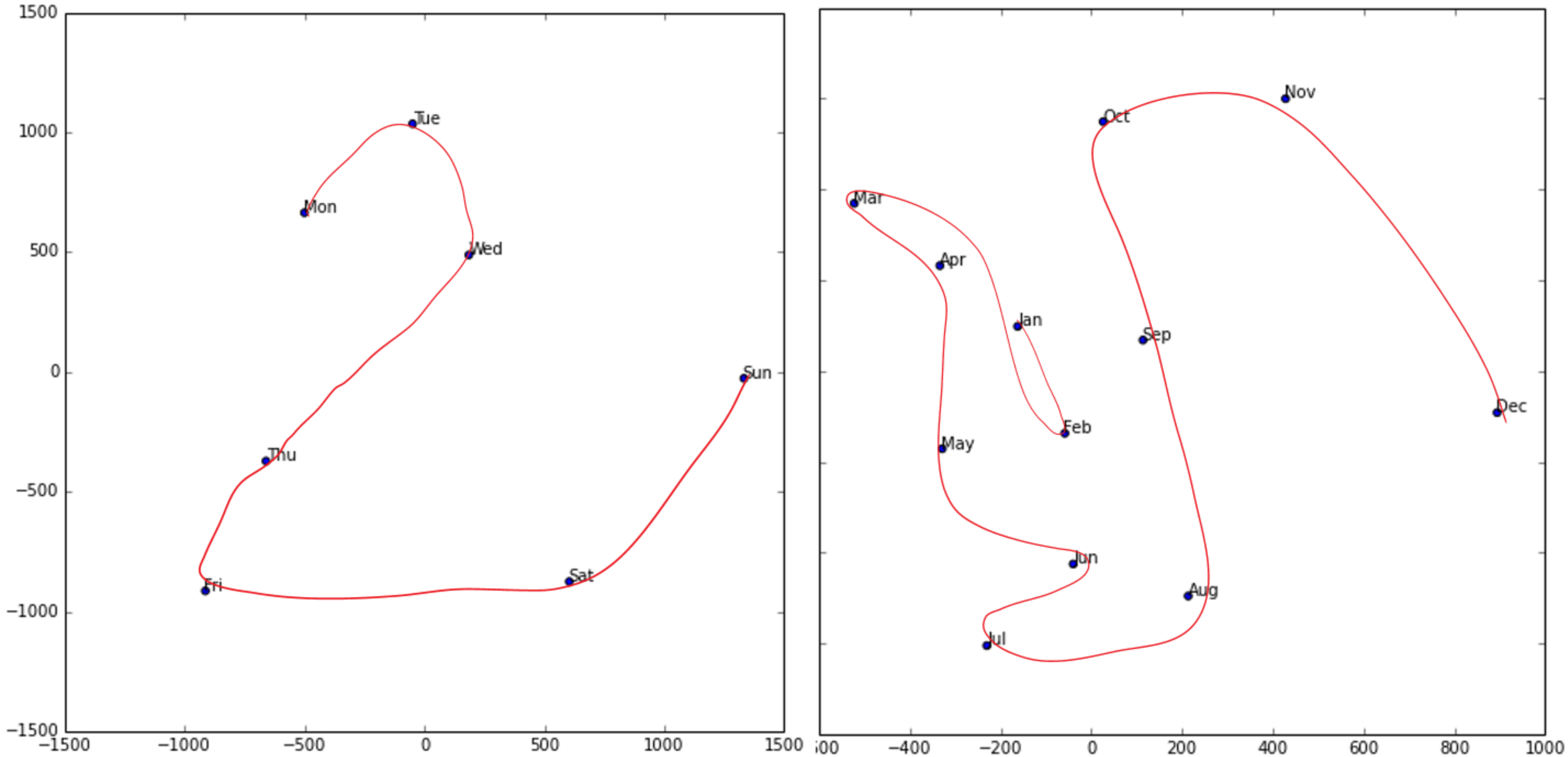




- 下图为实际地理距离和对应嵌入向量在制成的二维平面图图中距离的关系：两者基本呈现正相关，分布较为规律集中。



- 同样下图分别给出了一周7天和一年12个月份对应的嵌入向量二维平面图，我们也可在其中发现类似的规律



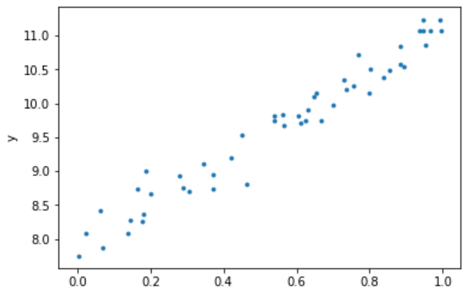
随机梯度下降

- 在lesson 1和lesson 5的基础上，此次更侧重于代码方面的讲述

In [2]:
```
# Here we generate some fake data
def lin(a,b,x): return a*x+b

def gen_fake_data(n, a, b):
    x = s = np.random.uniform(0,1,n)
    y = lin(a,b,x) + 0.1 * np.random.normal(0,3,n)
    return x, y

x, y = gen_fake_data(50, 3., 8.)
```

构建数据集

In [3]:
```
plt.scatter(x,y, s=8); plt.xlabel("x"); plt.ylabel("y");
```



In [4]:
```
def mse(y_hat, y): return ((y_hat - y) ** 2).mean()
```

Suppose we believe $a = 10$ and $b = 5$ then we can compute `y_hat` which is our *prediction* and then compute our error.

In [5]:
```
y_hat = lin(10,5,x)
mse(y_hat, y)
```

Out[5]: 4.8894873359935875

定义损失函数

In [6]:
```
def mse_loss(a, b, x, y): return mse(lin(a,b,x), y)
```

In [7]:
```
mse_loss(10, 5, x, y)
```

Out[7]: 4.8894873359935875

使用PyTorch实现

```
In [8]: # generate some more data
        x, y = gen_fake_data(10000, 3., 8.)
        x.shape, y.shape

Out[8]: ((10000,), (10000,))

In [9]: x,y = V(x),V(y)

In [10]: # Create random weights a and b, and wrap them in Variables.
         a = V(np.random.randn(1), requires_grad=True)
         b = V(np.random.randn(1), requires_grad=True)
         a,b
```

定义a，b参数

```
In [11]: learning_rate = 1e-3
         for t in range(10000):
             # Forward pass: compute predicted y using operations on Variables
             loss = mse_loss(a,b,x,y)
             if t % 1000 == 0: print(loss.data[0])

             # Computes the gradient of loss with respect to all Variables with requires_grad=True.
             # After this call a.grad and b.grad will be Variables holding the gradient
             # of the loss with respect to a and b respectively
             loss.backward()

             # Update a and b using gradient descent; a.data and b.data are Tensors,
             # a.grad and b.grad are Variables and a.grad.data and b.grad.data are Tensors
             a.data -= learning_rate * a.grad.data
             b.data -= learning_rate * b.grad.data

             # Zero the gradients
             a.grad.data.zero_()
             b.grad.data.zero_()

85.89212036132812
0.6416222453117371
0.10170342028141022
0.09614070504903793
0.09444241225719452
0.0931522473692894
0.0921601802110672
0.09139731526374817
0.09080982953310013
0.09035768359899521
```

更新a，b参数值

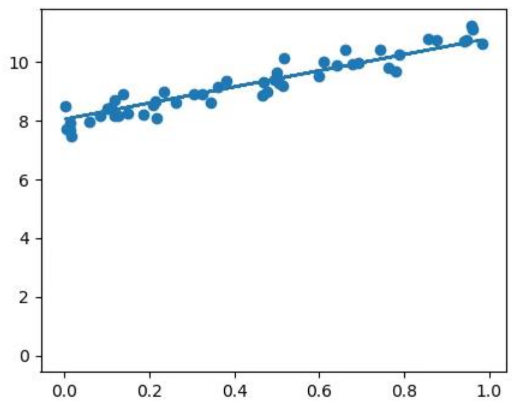
梯度趋0

使用Python实现

```
In [16]: x, y = gen_fake_data(50, 3., 8.)

In [17]: a_guess,b_guess = -1., 1.
         mse_loss(a_guess, b_guess, x, y)

Out[17]: 76.57995045535999

In [18]: lr=0.01
         def upd():
             global a_guess, b_guess
             y_pred = lin(a_guess, b_guess, x)
             dydb = 2 * (y_pred - y)
             dyda = x*dydb
             a_guess -= lr*dyda.mean()
             b_guess -= lr*dydb.mean()
```

更新a，b参数

```
In [19]: fig = plt.figure(dpi=100, figsize=(5, 4))
         plt.scatter(x,y)
         line, = plt.plot(x,lin(a_guess,b_guess,x))
         plt.close()

         def animate(i):
             line.set_ydata(lin(a_guess,b_guess,x))
             for i in range(30): upd()
             return line,

         ani = animation.FuncAnimation(fig, animate, np.arange(0, 20), interval=100)
         ani
```



循环神经网络（RNN）

RNN的四个优势：

- 可变长度序列
- 长时间相关性问题
- 状态表示能力
- 存储能力

Variable length sequence

Long-term dependency

Stateful Representation

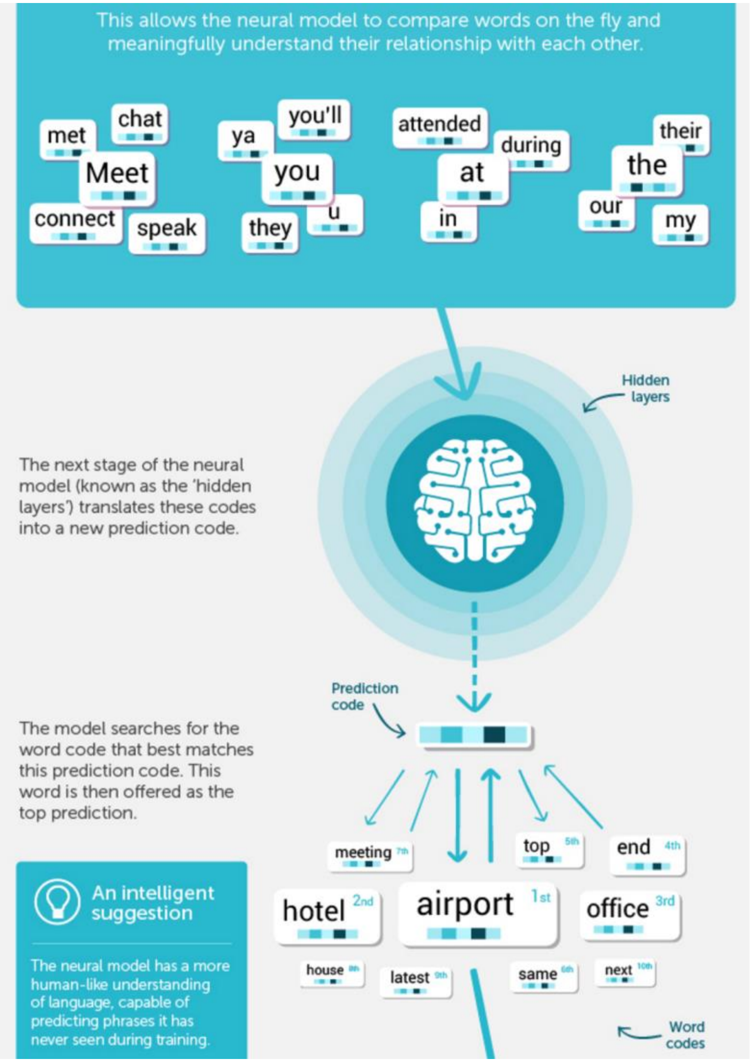
Memory

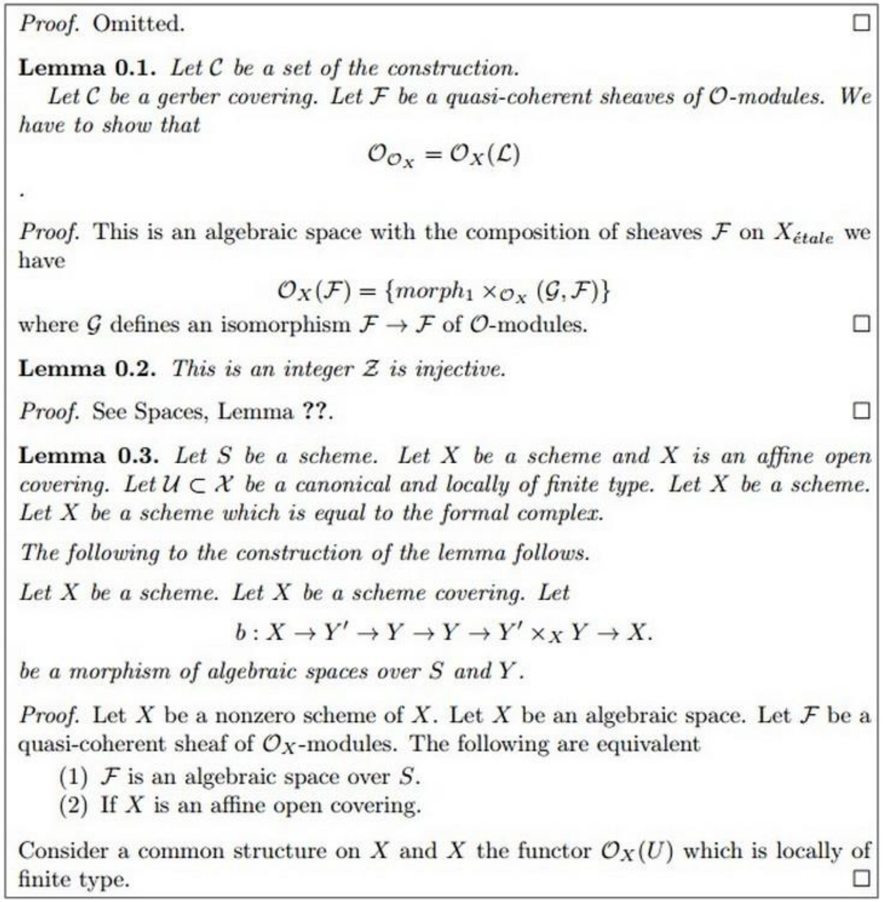
“I went to Nepal in 2009”

“In 2009, I went to Nepal”

```
<% comment do %>
  <div class="field">
    <%= f.label :title %><br>
    <%= f.text_field :title %>
  </div>
<% comment end %>
```

- 下图为RNN的应用实例





Proof. Omitted. □

Lemma 0.1. *Let $\mathcal{C}$ be a set of the construction.*

Let $\mathcal{C}$ be a gerber covering. Let $\mathcal{F}$ be a quasi-coherent sheaves of $\mathcal{O}$-modules. We have to show that

$$\mathcal{O}_{\mathcal{O}_X} = \mathcal{O}_X(\mathcal{L})$$

.

Proof. This is an algebraic space with the composition of sheaves $\mathcal{F}$ on $X_{étale}$ we have

$$\mathcal{O}_X(\mathcal{F}) = \{morph_1 \times_{\mathcal{O}_X} (\mathcal{G}, \mathcal{F})\}$$

where $\mathcal{G}$ defines an isomorphism $\mathcal{F} \to \mathcal{F}$ of $\mathcal{O}$-modules. □

Lemma 0.2. *This is an integer $\mathcal{Z}$ is injective.*

Proof. See Spaces, Lemma ??. □

Lemma 0.3. *Let S be a scheme. Let X be a scheme and X is an affine open covering. Let $\mathcal{U} \subset \mathcal{X}$ be a canonical and locally of finite type. Let X be a scheme. Let X be a scheme which is equal to the formal complex.*

The following to the construction of the lemma follows.

Let X be a scheme. Let X be a scheme covering. Let

$$b : X \to Y' \to Y \to Y \to Y' \times_X Y \to X.$$

be a morphism of algebraic spaces over S and Y.

Proof. Let X be a nonzero scheme of X. Let X be an algebraic space. Let $\mathcal{F}$ be a quasi-coherent sheaf of $\mathcal{O}_X$-modules. The following are equivalent

(1) $\mathcal{F}$ is an algebraic space over S.
(2) If X is an affine open covering.

Consider a common structure on X and X the functor $\mathcal{O}_X(U)$ which is locally of finite type. □

\begin{proof} We may assume that $\mathcal{I}$ is an abelian sheaf on $\mathcal{C}$. \item Given a morphism $\Delta : \mathcal{F} \to \mathcal{I}$

上海交通大学
Shanghai Jiao Tong University

Basic NN with single hidden layer

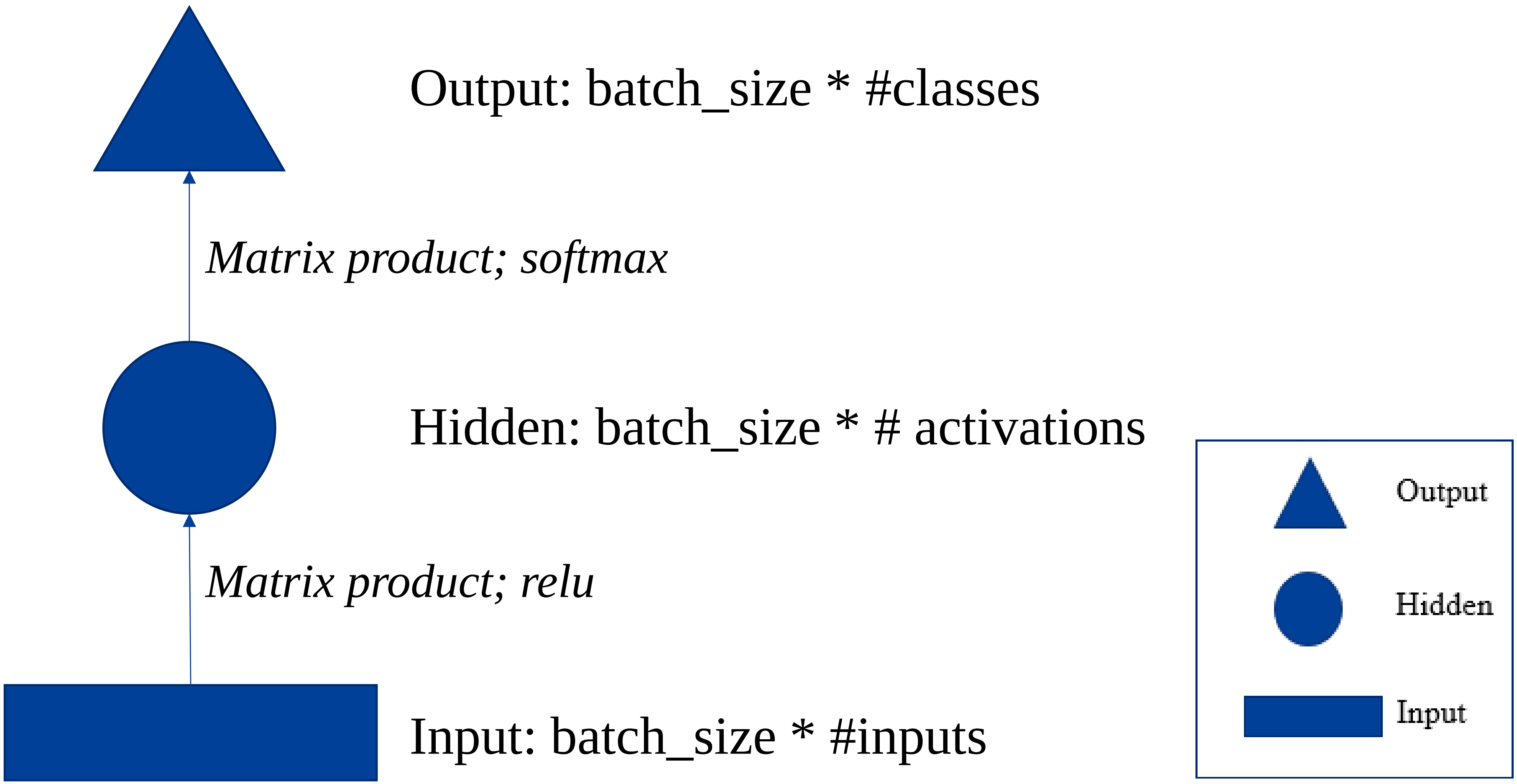
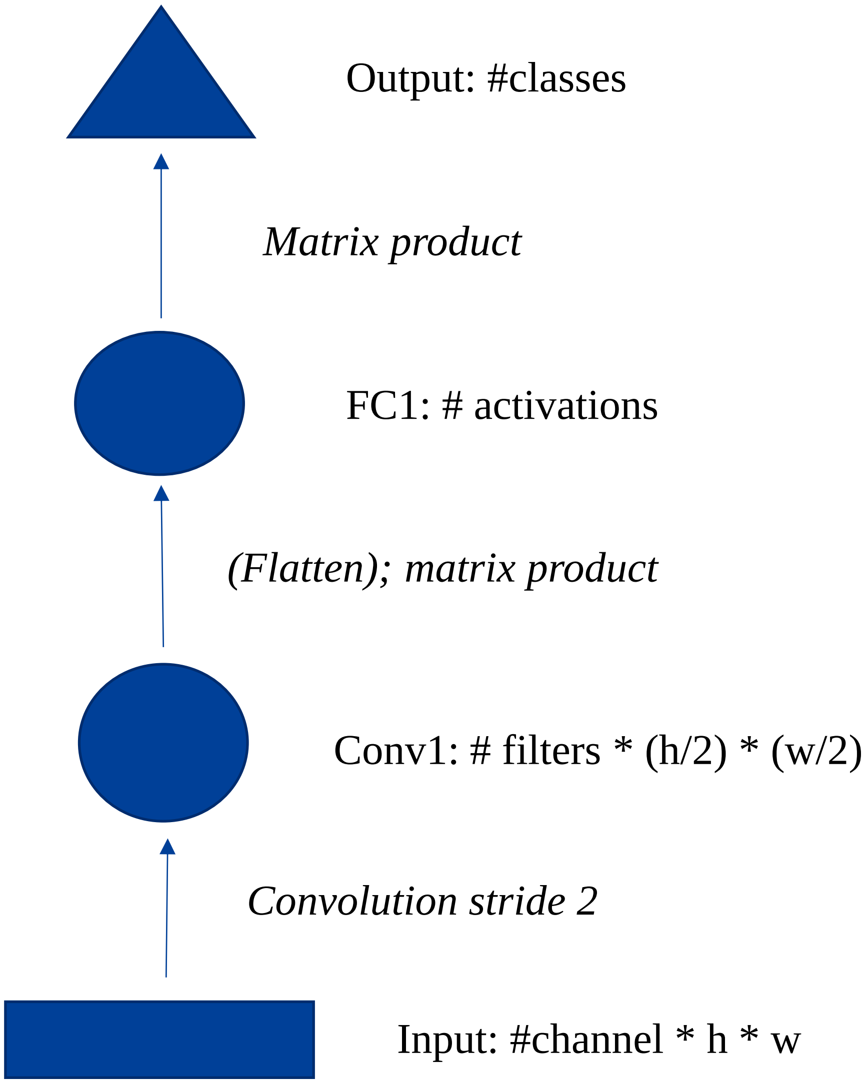
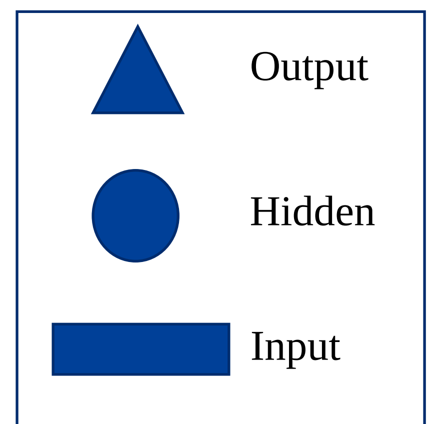


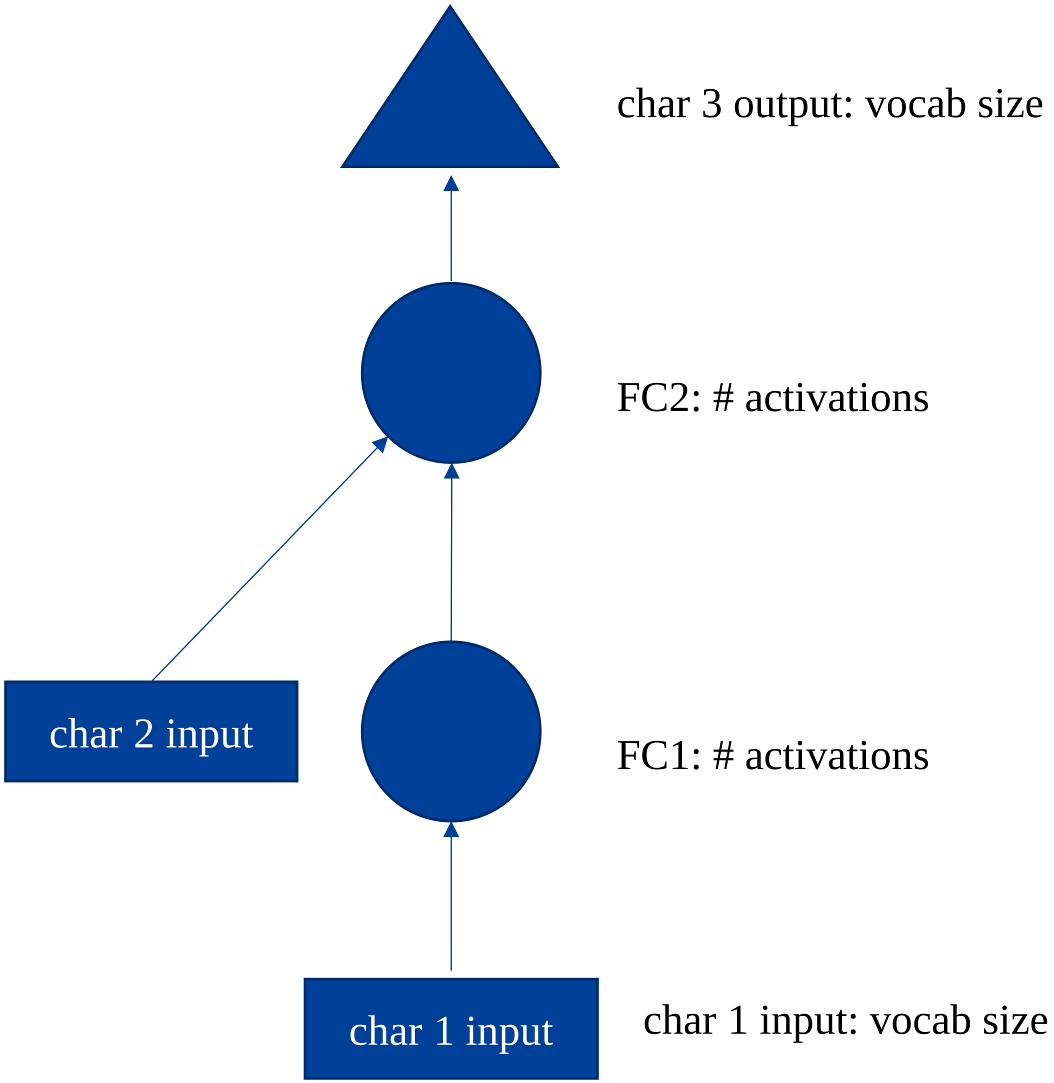
Image CNN with single dense hidden layer



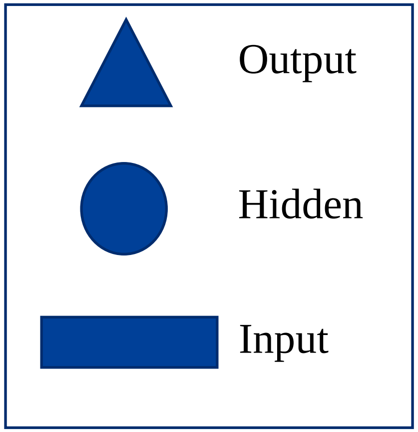
NB: batch_size dimension and activation function not shown here or in following slides



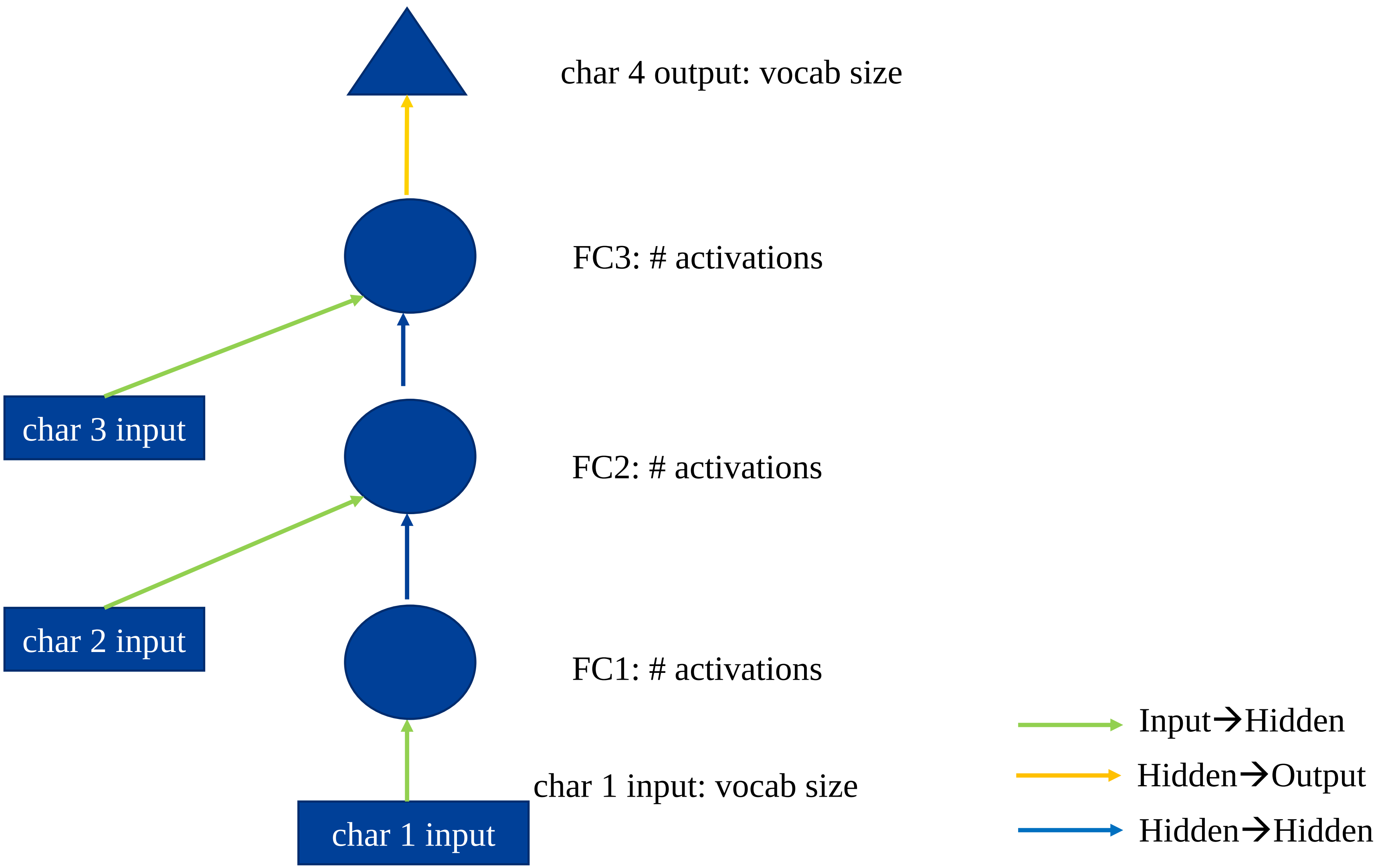
Predicting char 3 using chars 1 & 2



NB: layer operations not shown; remember that arrows represent layer operations



Predicting char 4 using chars 1, 2 & 3



Predicting char 4 using chars 1, 2 & 3

In [5]:
```
chars = sorted(list(set(text)))
vocab_size = len(chars)+1
print('total chars:', vocab_size)
```

```
total chars: 85
```

Sometimes it's useful to have a zero value in the dataset, e.g. for padding

In [6]:
```
chars.insert(0, "\0")

''.join(chars[1:-6])
```

Out[6]:
```
'\n !"\'(),-.0123456789:;=?ABCDEFGHIJKLMNOPQRSTUVWXYZ[]_abcdefghijklmnopqrstuvwxy'
```

Map from chars to indices and back again

In [7]:
```
char_indices = {c: i for i, c in enumerate(chars)}
indices_char = {i: c for i, c in enumerate(chars)}
```

定义id与字符转换函数

idx will be the data we use from now on - it simply converts all the characters to their index (based on the mapping above)

In [8]:
```
idx = [char_indices[c] for c in text]

idx[:10]
```

Out[8]:
```
[40, 42, 29, 30, 25, 27, 29, 1, 1, 1]
```

In [10]:
```
cs=3
c1_dat = [idx[i]   for i in range(0, len(idx)-cs, cs)]
c2_dat = [idx[i+1] for i in range(0, len(idx)-cs, cs)]
c3_dat = [idx[i+2] for i in range(0, len(idx)-cs, cs)]
c4_dat = [idx[i+3] for i in range(0, len(idx)-cs, cs)]
```

以cs=3为步长

Our inputs

In [11]:
```
x1 = np.stack(c1_dat)
x2 = np.stack(c2_dat)
x3 = np.stack(c3_dat)
```

Our output

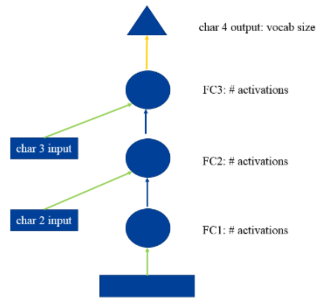
In [14]:
```
y = np.stack(c4_dat)
```

The first 4 inputs and outputs

In [14]:
```
x1[:4], x2[:4], x3[:4]
```

Out[14]:
```
(array([40, 30, 29,  1]), array([42, 25,  1, 43]), array([29, 27,  1, 45]))
```

Predicting char 4 using chars 1, 2 & 3



Predicting char 4 using chars 1, 2 & 3

```
In [17]: n_hidden = 256
```

The number of latent factors to create (i.e. the size of the embedding matrix)

```
In [18]: n_fac = 42
```

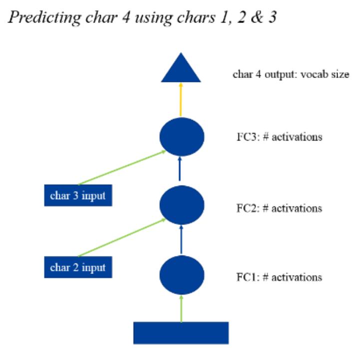
```
In [19]: class Char3Model(nn.Module):
             def __init__(self, vocab_size, n_fac):
                 super().__init__()
                 self.e = nn.Embedding(vocab_size, n_fac)

                 # The 'green arrow' from our diagram - the layer operation from input to hidden
                 self.l_in = nn.Linear(n_fac, n_hidden)

                 # The 'orange arrow' from our diagram - the layer operation from hidden to hidden
                 self.l_hidden = nn.Linear(n_hidden, n_hidden)

                 # The 'blue arrow' from our diagram - the layer operation from hidden to output
                 self.l_out = nn.Linear(n_hidden, vocab_size)

             def forward(self, c1, c2, c3):
                 in1 = F.relu(self.l_in(self.e(c1)))
                 in2 = F.relu(self.l_in(self.e(c2)))
                 in3 = F.relu(self.l_in(self.e(c3)))

                 h = V(torch.zeros(in1.size()).cuda())
                 h = F.tanh(self.l_hidden(h+in1))
                 h = F.tanh(self.l_hidden(h+in2))
                 h = F.tanh(self.l_hidden(h+in3))

                 return F.log_softmax(self.l_out(h))
```

激活函数：tanh(x)

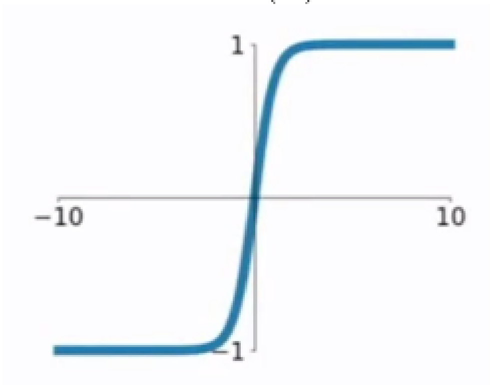
```
In [20]: md = ColumnarModelData.from_arrays('.', [-1], np.stack([x1,x2,x3], axis=1), y, bs=512)

In [21]: m = Char3Model(vocab_size, n_fac).cuda()

In [22]: it = iter(md.trn_dl)
         *xs,yt = next(it)
         t = m(*V(xs))

In [23]: opt = optim.Adam(m.parameters(), 1e-2)

In [24]: fit(m, md, 1, opt, F.nll_loss)
```

Epoch 100% 1/1 [00:02<00:00, 2.26s/it]

```
epoch    trn_loss   val_loss
    0    2.081694   0.689759

Out[24]: [array([0.68976])]
```

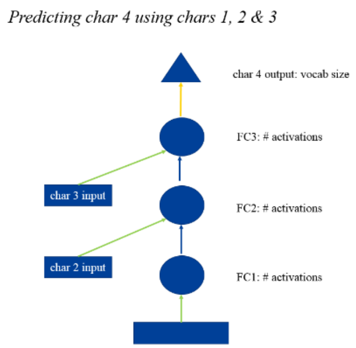


$$tanh(x) = \frac{sinh(x)}{cosh(x)} = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$



相较于sigmoid函数，其分布在（-1,1）以0为中心，有更好的权值更新效率

Predicting char 4 using chars 1, 2 & 3




```
In [25]: set_lrs(opt, 0.001)
```

```
In [26]: fit(m, md, 1, opt, F.nll_loss)
```

Epoch 100% 1/1 [00:02<00:00, 2.14s/it]

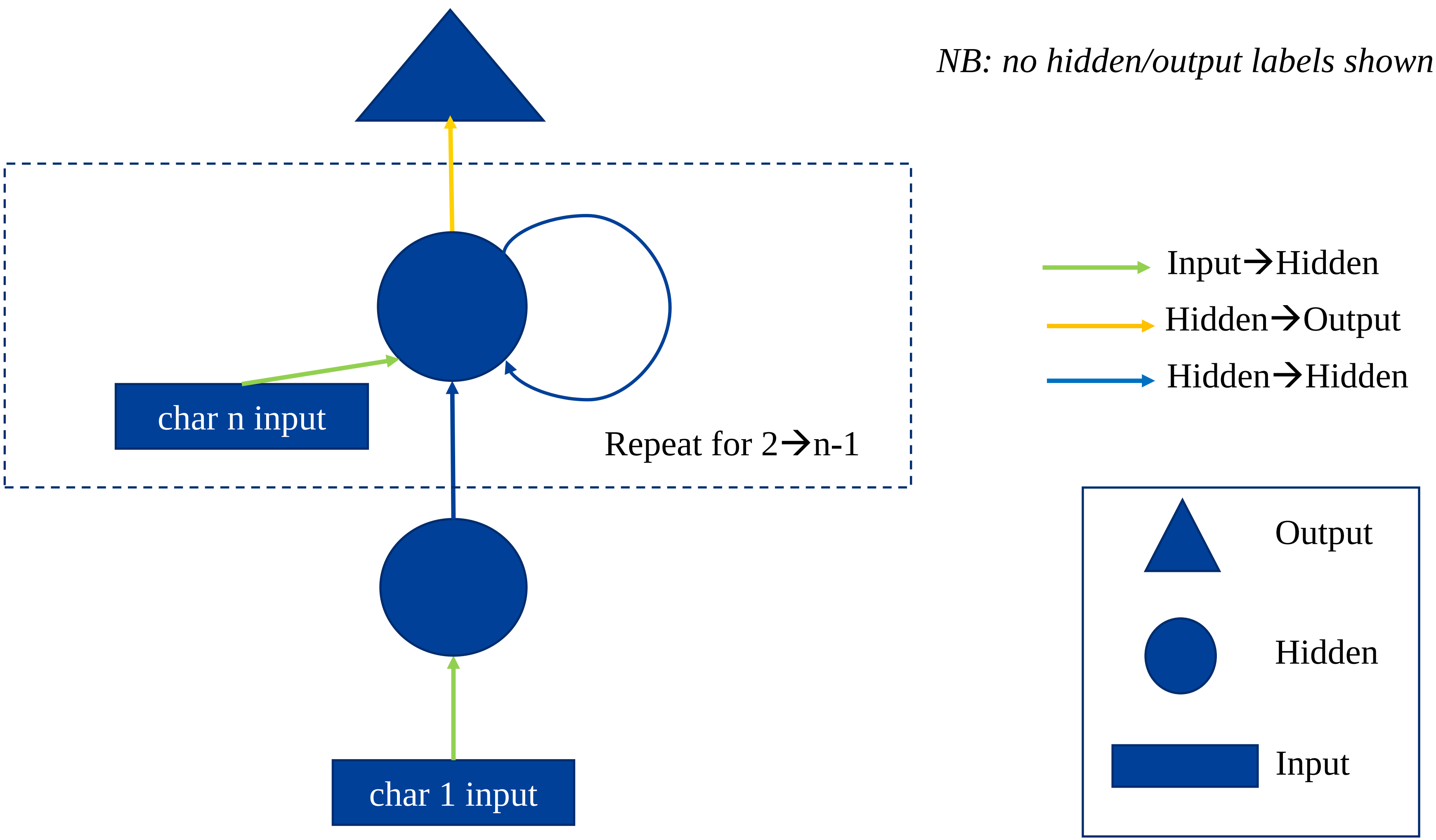
```
epoch      trn_loss   val_loss
    0      1.826699   0.636193
```

Test model

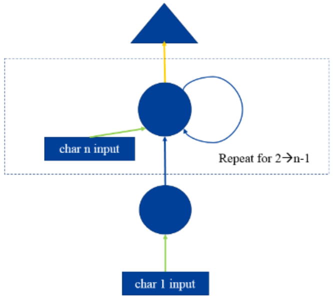
```
In [27]: def get_next(inp):
             idxs = T(np.array([char_indices[c] for c in inp]))
             p = m(*VV(idxs))
             i = np.argmax(to_np(p))
             return chars[i]
```

```
In [28]: get_next('y. ')
Out[28]: 'T'

In [29]: get_next('ppl')
Out[29]: 'e'

In [30]: get_next(' th')
Out[30]: 'e'

In [31]: get_next('and')
Out[31]: ' '
```

Predicting char n using chars 1 to n-1



RNN



In [36]:
```
cs=8
```

For each of 0 through 7, create a list of every 8th character with that starting point. These will be the 8 inputs to our model.

In [37]:
```
c_in_dat = [[idx[i+j] for i in range(cs)] for j in range(len(idx)-cs)]
```

步长为1

Then create a list of the next character in each of these series. This will be the labels for our model.

In [38]:
```
c_out_dat = [idx[j+cs] for j in range(len(idx)-cs)]
```

In [39]:
```
xs = np.stack(c_in_dat, axis=0)
```

In [40]:
```
xs.shape
```

Out[40]:
```
(600885, 8)
```

In [41]:
```
y = np.stack(c_out_dat)
```

In [42]:
```
xs[:cs,:cs]
```

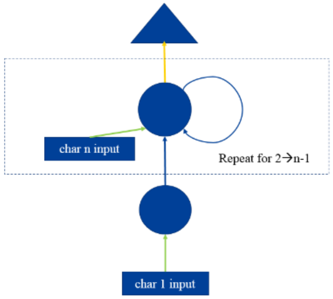
Out[42]:
```
array([[40, 42, 29, 30, 25, 27, 29,  1],
       [42, 29, 30, 25, 27, 29,  1,  1],
       [29, 30, 25, 27, 29,  1,  1,  1],
       [30, 25, 27, 29,  1,  1,  1, 43],
       [25, 27, 29,  1,  1,  1, 43, 45],
       [27, 29,  1,  1,  1, 43, 45, 40],
       [29,  1,  1,  1, 43, 45, 40, 40],
       [ 1,  1,  1, 43, 45, 40, 40, 39]])
```

...and this is the next character after each sequence.

In [43]:
```
y[:cs]
```

Out[43]:
```
array([ 1,  1, 43, 45, 40, 40, 39, 43])
```

RNN




```
In [44]: val_idx = get_cv_idxs(len(idx)-cs-1)

In [45]: md = ColumnarModelData.from_arrays('.', val_idx, xs, y, bs=512)

In [46]: class CharLoopModel(nn.Module):
             # This is an RNN!
             def __init__(self, vocab_size, n_fac):
                 super().__init__()
                 self.e = nn.Embedding(vocab_size, n_fac)
                 self.l_in = nn.Linear(n_fac, n_hidden)
                 self.l_hidden = nn.Linear(n_hidden, n_hidden)
                 self.l_out = nn.Linear(n_hidden, vocab_size)

             def forward(self, *cs):
                 bs = cs[0].size(0)
                 h = V(torch.zeros(bs, n_hidden).cuda())
                 for c in cs:
                     inp = F.relu(self.l_in(self.e(c)))
                     h = F.tanh(self.l_hidden(h+inp))

                 return F.log_softmax(self.l_out(h), dim=-1)
```

信息损失

```
In [47]: m = CharLoopModel(vocab_size, n_fac).cuda()
         opt = optim.Adam(m.parameters(), 1e-2)

In [48]: fit(m, md, 1, opt, F.nll_loss)
```

Epoch 100% 1/1 [00:12<00:00, 12.41s/it]

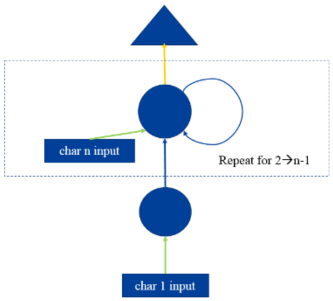
```
epoch      trn_loss   val_loss
    0      2.055552   2.053333

Out[48]: [array([2.05333])]
```

```
In [51]: class CharLoopConcatModel(nn.Module):
             def __init__(self, vocab_size, n_fac):
                 super().__init__()
                 self.e = nn.Embedding(vocab_size, n_fac)
                 self.l_in = nn.Linear(n_fac+n_hidden, n_hidden)
                 self.l_hidden = nn.Linear(n_hidden, n_hidden)
                 self.l_out = nn.Linear(n_hidden, vocab_size)

             def forward(self, *cs):
                 bs = cs[0].size(0)
                 h = V(torch.zeros(bs, n_hidden).cuda())
                 for c in cs:
                     inp = torch.cat((h, self.e(c)), 1)
                     inp = F.relu(self.l_in(inp))
                     h = F.tanh(self.l_hidden(inp))

                 return F.log_softmax(self.l_out(h), dim=-1)

In [52]: m = CharLoopConcatModel(vocab_size, n_fac).cuda()
         opt = optim.Adam(m.parameters(), 1e-3)

In [53]: it = iter(md.trn_dl)
         *xs,yt = next(it)
         t = m(*V(xs))

In [54]: fit(m, md, 1, opt, F.nll_loss)
```

Epoch 100% 1/1 [00:12<00:00, 12.65s/it]

```
epoch      trn_loss   val_loss
    0      1.837322   1.804712

Out[54]: [array([1.80471])]
```

RNN with pytorch




```
In [61]: class CharRnn(nn.Module):
             def __init__(self, vocab_size, n_fac):
                 super().__init__()
                 self.e = nn.Embedding(vocab_size, n_fac)
                 self.rnn = nn.RNN(n_fac, n_hidden)
                 self.l_out = nn.Linear(n_hidden, vocab_size)

             def forward(self, *cs):
                 bs = cs[0].size(0)
                 h = V(torch.zeros(1, bs, n_hidden))
                 inp = self.e(torch.stack(cs))
                 outp,h = self.rnn(inp, h)

                 return F.log_softmax(self.l_out(outp[-1]), dim=-1)
```

```
In [62]: m = CharRnn(vocab_size, n_fac).cuda()
         opt = optim.Adam(m.parameters(), 1e-3)
```

```
In [63]: it = iter(md.trn_dl)
         *xs,yt = next(it)
```

```
In [64]: t = m.e(V(torch.stack(xs)))
         t.size()

Out[64]: torch.Size([8, 512, 42])
```

```
In [65]: ht = V(torch.zeros(1, 512,n_hidden))
         outp, hn = m.rnn(t, ht)
         outp.size(), hn.size()

Out[65]: (torch.Size([8, 512, 256]), torch.Size([1, 512, 256]))
```

```
In [66]: t = m(*V(xs)); t.size()

Out[66]: torch.Size([512, 85])
```

```
In [67]: fit(m, md, 4, opt, F.nll_loss)
```

Epoch 100% 4/4 [00:46<00:00, 11.57s/it]

epoch	trn_loss	val_loss
0	1.86082	1.837214
1	1.676803	1.668071
2	1.574662	1.585919
3	1.535512	1.546742

```
Out[67]: [array([1.54674])]
```

Test model

```
In [117]: def get_next(inp):
              idxs = T(np.array([char_indices[c] for c in inp]))
              p = m(*VV(idxs))
              i = np.argmax(to_np(p))
              return chars[i]
```

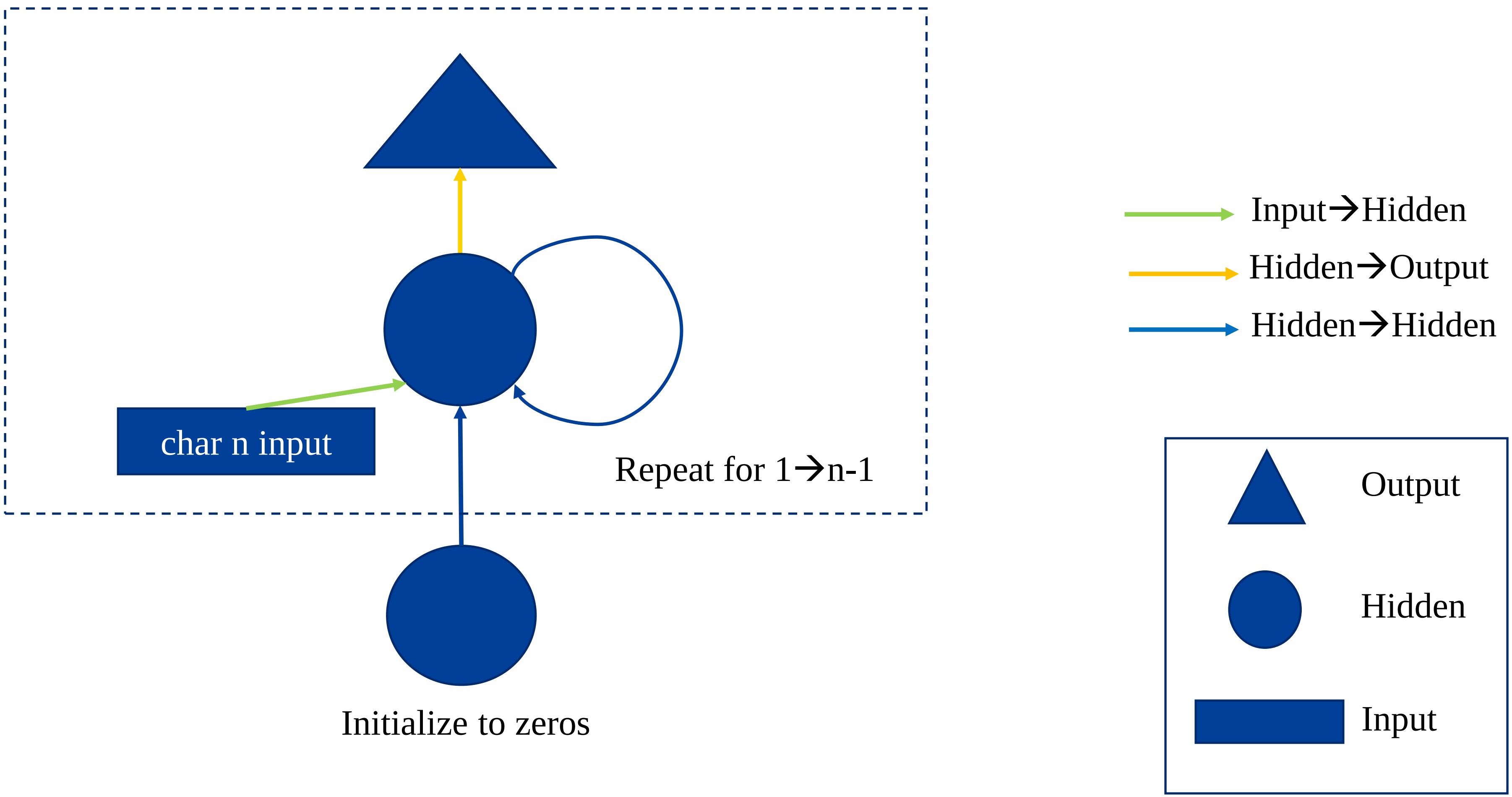
```
In [118]: get_next('for thos')

Out[118]: 'e'
```

```
In [119]: def get_next_n(inp, n):
              res = inp
              for i in range(n):
                  c = get_next(inp)
                  res += c
                  inp = inp[1:]+c
              return res
```

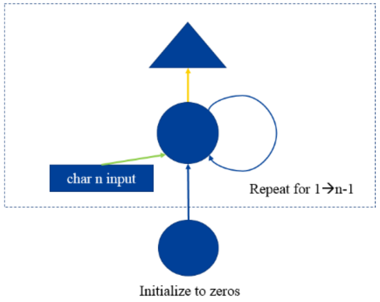
```
In [120]: get_next_n('for thos', 40)

Out[120]: 'for those the same the same the same the same th'
```

Predicting chars 2 to n using chars 1 to n-1

NB: no hidden/output labels shown



Multi-output model



In [78]:
```
c_in_dat = [[idx[i+j] for i in range(cs)] for j in range(0, len(idx)-cs-1, cs)]
```

Then create the exact same thing, offset by 1, as our labels

以cs为步长

In [79]:
```
c_out_dat = [[idx[i+j] for i in range(cs)] for j in range(1, len(idx)-cs, cs)]
```

In [80]:
```
xs = np.stack(c_in_dat)
xs.shape
```

Out[80]:
```
(75111, 8)
```

In [81]:
```
ys = np.stack(c_out_dat)
ys.shape
```

Out[81]:
```
(75111, 8)
```

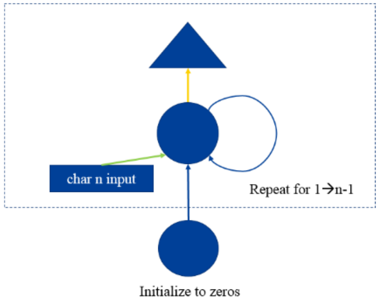
In [82]:
```
xs[:cs,:cs]
```

Out[82]:
```
array([[40, 42, 29, 30, 25, 27, 29,  1],
       [ 1,  1, 43, 45, 40, 40, 39, 43],
       [33, 38, 31,  2, 73, 61, 54, 73],
       [ 2, 44, 71, 74, 73, 61,  2, 62],
       [72,  2, 54,  2, 76, 68, 66, 54],
       [67,  9,  9, 76, 61, 54, 73,  2],
       [73, 61, 58, 67, 24,  2, 33, 72],
       [ 2, 73, 61, 58, 71, 58,  2, 67]])
```

In [83]:
```
ys[:cs,:cs]
```

Out[83]:
```
array([[42, 29, 30, 25, 27, 29,  1,  1],
       [ 1, 43, 45, 40, 40, 39, 43, 33],
       [38, 31,  2, 73, 61, 54, 73,  2],
       [44, 71, 74, 73, 61,  2, 62, 72],
       [ 2, 54,  2, 76, 68, 66, 54, 67],
       [ 9,  9, 76, 61, 54, 73,  2, 73],
       [61, 58, 67, 24,  2, 33, 72,  2],
       [73, 61, 58, 71, 58,  2, 67, 68]])
```

Multi-output model



Create and train model

In [84]:
```
val_idx = get_cv_idxs(len(xs)-cs-1)
```

In [85]:
```
md = ColumnarModelData.from_arrays('.', val_idx, xs, ys, bs=512)
```

In [86]:
```
class CharSeqRnn(nn.Module):
    def __init__(self, vocab_size, n_fac):
        super().__init__()
        self.e = nn.Embedding(vocab_size, n_fac)
        self.rnn = nn.RNN(n_fac, n_hidden)
        self.l_out = nn.Linear(n_hidden, vocab_size)

    def forward(self, *cs):
        bs = cs[0].size(0)
        h = V(torch.zeros(1, bs, n_hidden))
        inp = self.e(torch.stack(cs))
        outp,h = self.rnn(inp, h)
        return F.log_softmax(self.l_out(outp), dim=-1)
```

In [87]:
```
m = CharSeqRnn(vocab_size, n_fac).cuda()
opt = optim.Adam(m.parameters(), 1e-3)
```

In [88]:
```
it = iter(md.trn_dl)
*xst,yt = next(it)
```

In [89]:
```
def nll_loss_seq(inp, targ):
    sl,bs,nh = inp.size()
    targ = targ.transpose(0,1).contiguous().view(-1)
    return F.nll_loss(inp.view(-1,nh), targ)
```

In [90]:
```
fit(m, md, 4, opt, nll_loss_seq)
```

Epoch 100% 4/4 [00:05<00:00, 1.46s/it]

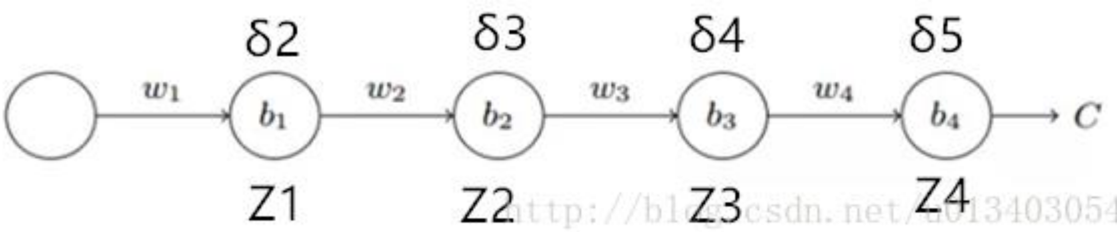
epoch	trn_loss	val_loss
0	2.603039	2.414561
1	2.294401	2.205506
2	2.143854	2.09426
3	2.049728	2.017031

Out[90]:
```
[array([2.01703])]
```

Identity initialization

深度神经网络训练的时候，采用的是反向传播方式，该方式使用链式求导，计算每层梯度的时候会涉及一些连乘操作，因此如果网络过深：

- 若连乘的因子大部分小于1，最后乘积的结果可能趋于0，也就是梯度消失，导致网络层的参数无法更新
- 若连乘的因子大部分大于1，最后乘积可能趋于无穷，这就是梯度爆炸




```
In [93]: m = CharSeqRnn(vocab_size, n_fac).cuda()
         opt = optim.Adam(m.parameters(), 1e-2)

In [94]: m.rnn.weight_hh_l0.data.copy_(torch.eye(n_hidden))

Out[94]:
    1     0     0  ...     0     0     0
    0     1     0  ...     0     0     0
    0     0     1  ...     0     0     0
      ...          ⋱          ...
    0     0     0  ...     1     0     0
    0     0     0  ...     0     1     0
    0     0     0  ...     0     0     1
[torch.cuda.FloatTensor of size 256x256 (GPU 0)]

In [95]: fit(m, md, 4, opt, nll_loss_seq)

         Epoch  100% 4/4 [00:05<00:00, 1.47s/it]

epoch      trn_loss   val_loss
    0      2.37629    2.226769
    1      2.100347   2.046417
    2      1.997875   1.970613
    3      1.947667   1.938284

Out[95]: [array([1.93828])]
```

- 由于任何矩阵与单位矩阵相乘均为其本身，我们可以将隐层之间的权值矩阵初始化为单位矩阵，能有效地避免上述问题
- 因此我们可以采用更高的学习率，加快计算速度，增强预测效果

$$A \cdot \begin{bmatrix} 1 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & 1 \end{bmatrix}_n = A$$

谢谢！