Python Libraries for Deep Learning with Sequences

Alex Rubinsteyn
PyData NYC 2015

HammerLab @ Mount Sinai













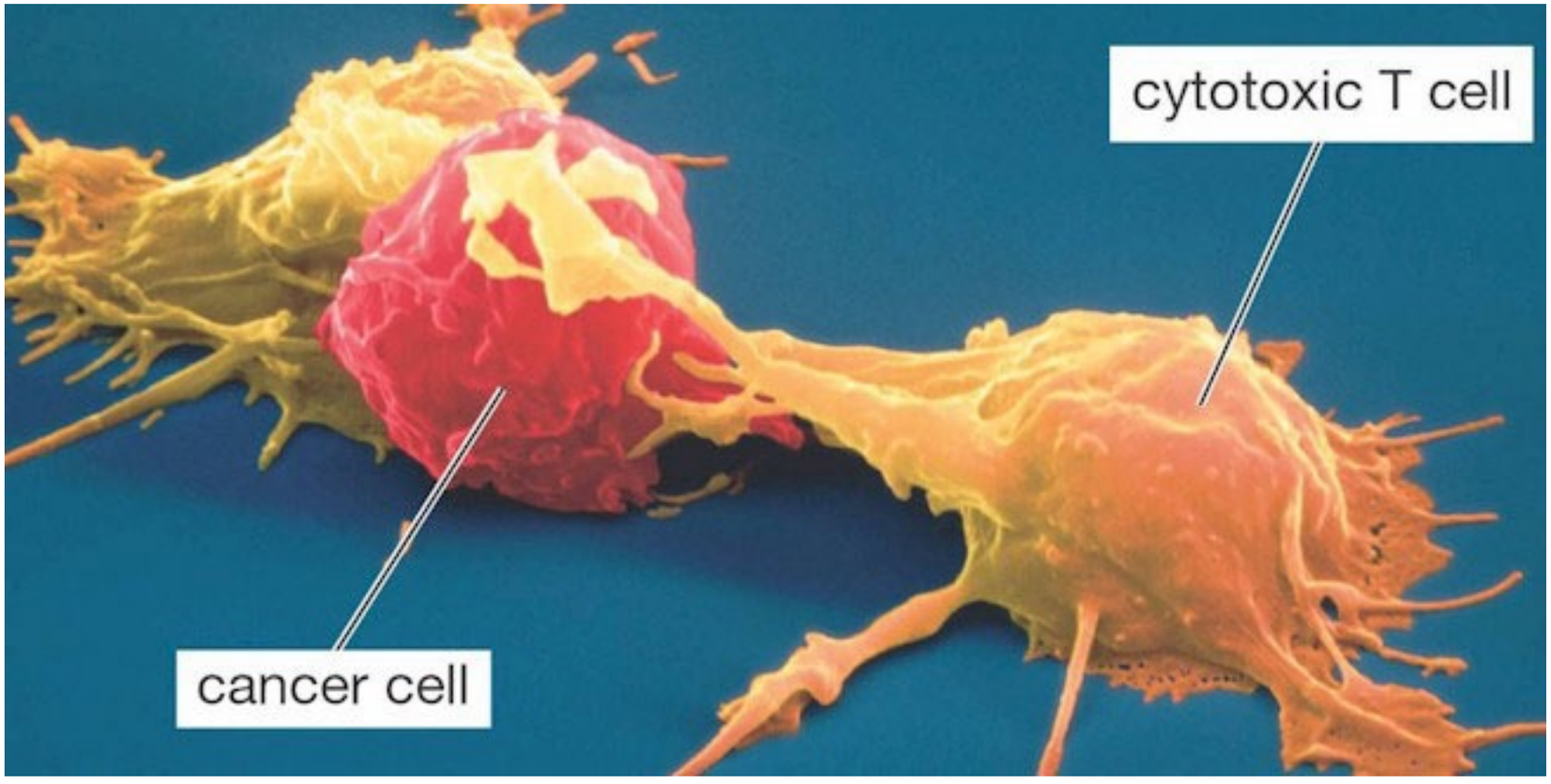

http://www.hammerlab.org/research/

Not biologists!

Most lab members have a background in Computer Science and/or Mathematics:

- Distributed data management
- Machine learning & statistics
- Programming languages
- Data visualization

Motivation: Cancer Immunology meets Machine Learning



T-cells recognize specific amino acid sequences presented on the surface of tumor cells.

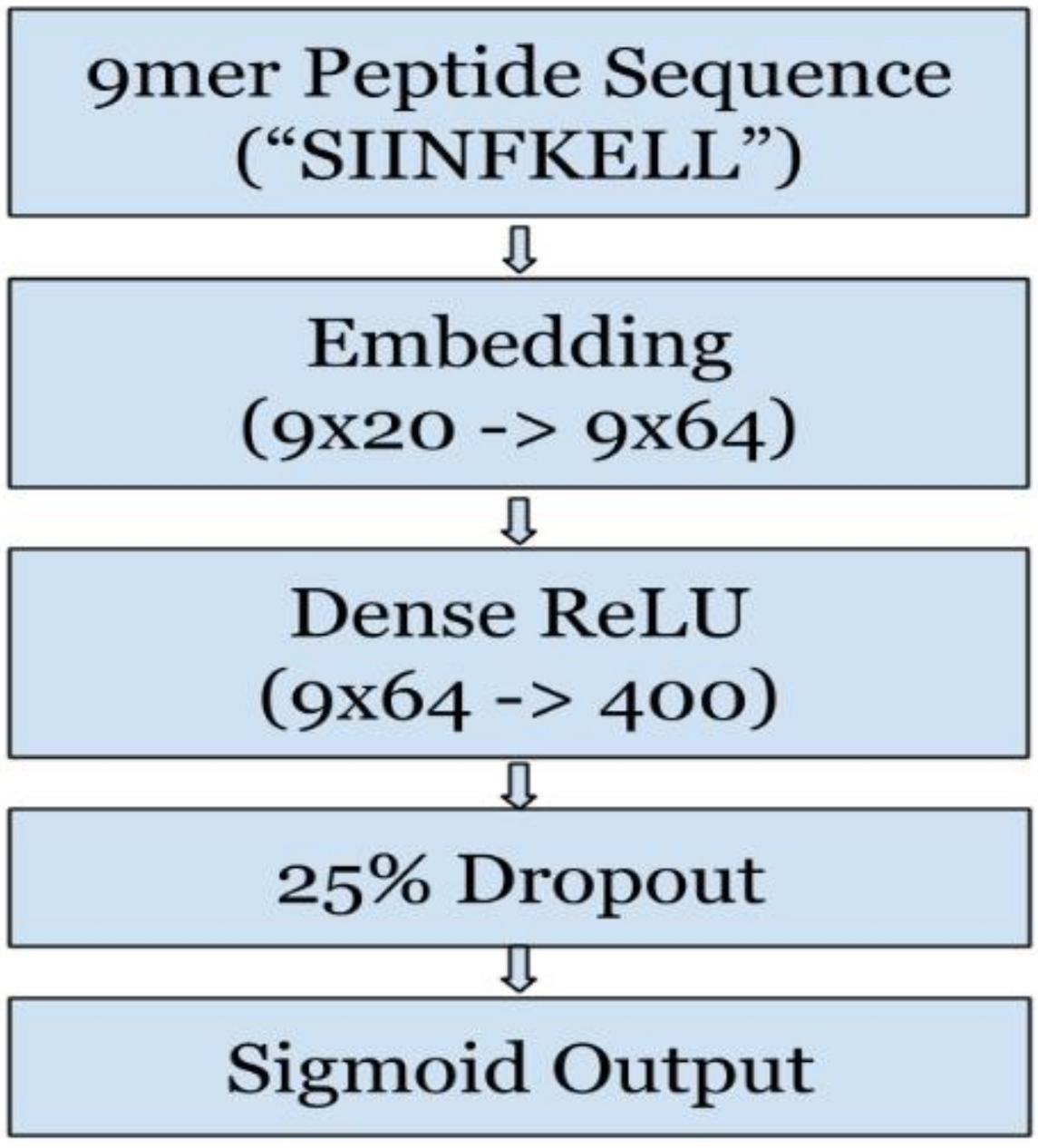
Motivation: Cancer Immunology meets Machine Learning

Q: Which mutated protein fragments will cause a targeted immune response against tumor cells?

- Protein fragments must be presented on the cell surface to be recognized by the immune system.
- Immunologists have collected ~300k examples of protein fragment "affinity" for the proteins that present them to the immune system.
- Build a model from protein fragment sequence to binding affinity! Amino acid sequence -> [0,1]

Learning with fixed length sequences is easy!

- 1-of-k encoding turns sequence of *n* characters into *k*n* length binary vector
- Vector embedding (e.g. word2vec) can learn better representations from the data directly.

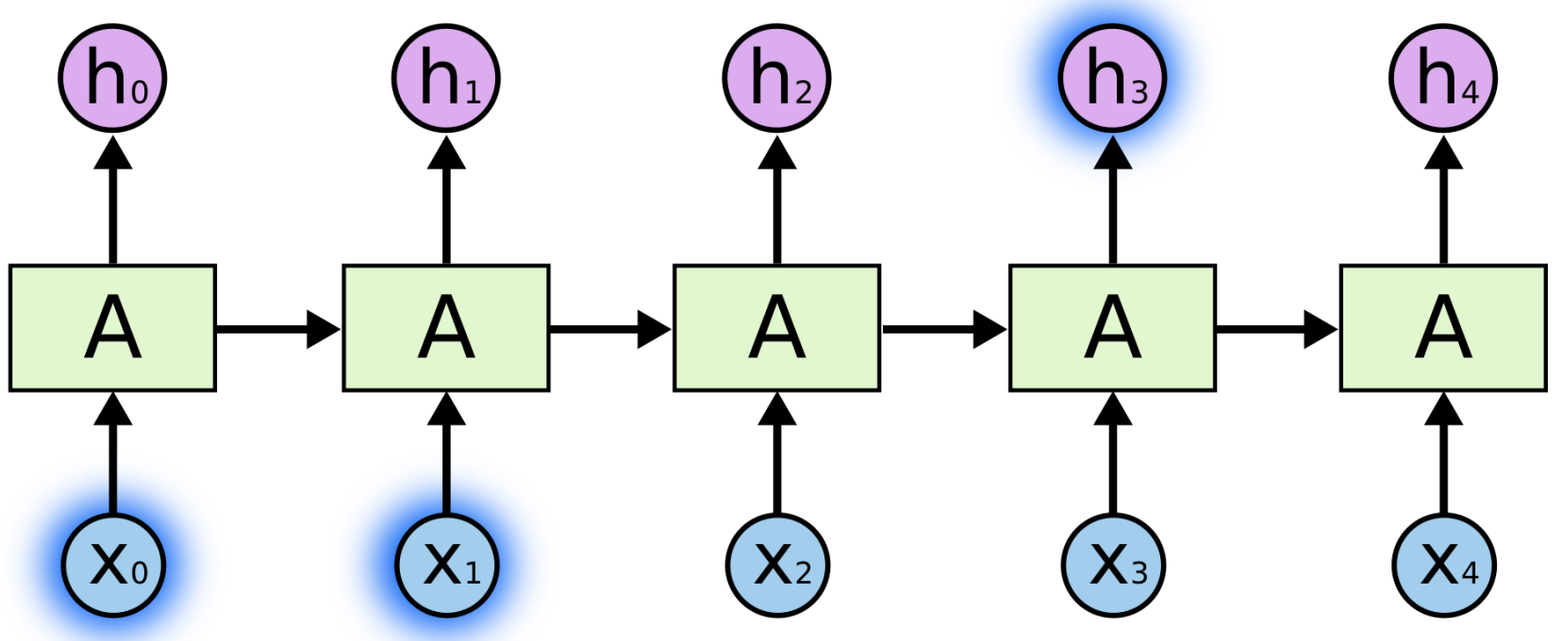


Varying length sequences are hard!

- (Common approach) *bag of words / n-grams:* Doesn't work when order actually matters!
- *"Artisanal" features & sliding windows*: how to combine varying numbers of windows?
- *Structured prediction (e.g. CRFs)*: work for specific tasks BUT non-compositional (hard to combine with NNs) & inference is often slow

Simple RNNs

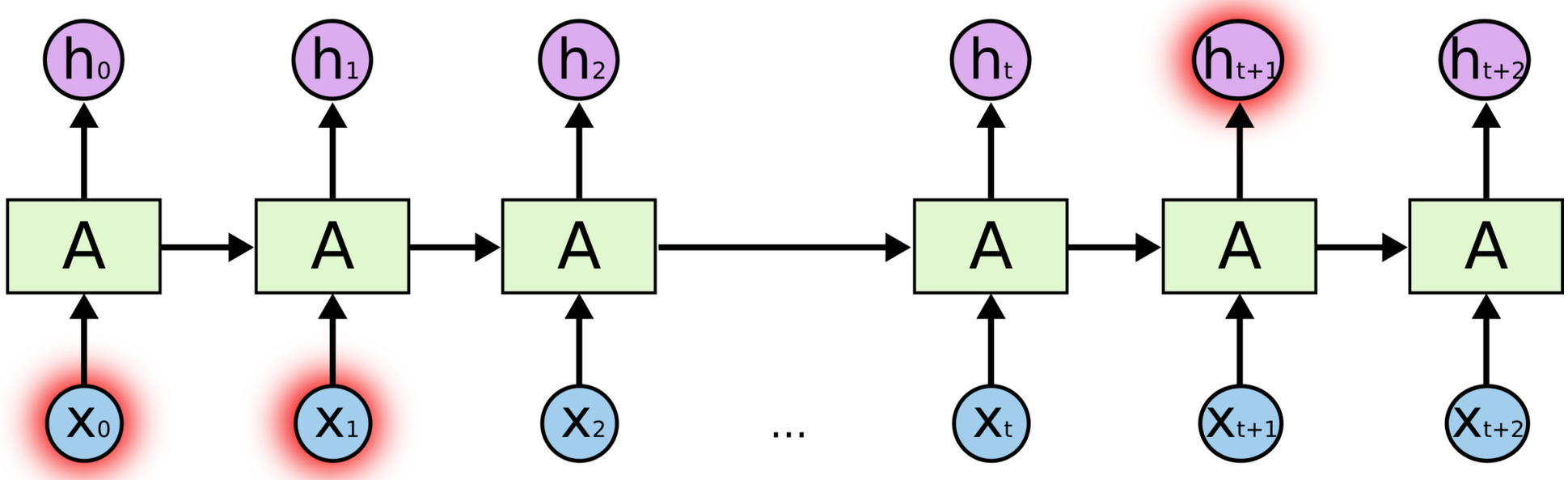
- Neural network with two kinds of inputs: input from current time step (X_t) & previous output (h_{t-1})
- Train using "Back-propagation through time" (BPTT)
- Works for short time-scale dependencies between inputs and outputs



Source: http://colah.github.io/posts/2015-08-Understanding-LSTMs/

Simple RNNs

- …but what happens across many time steps?

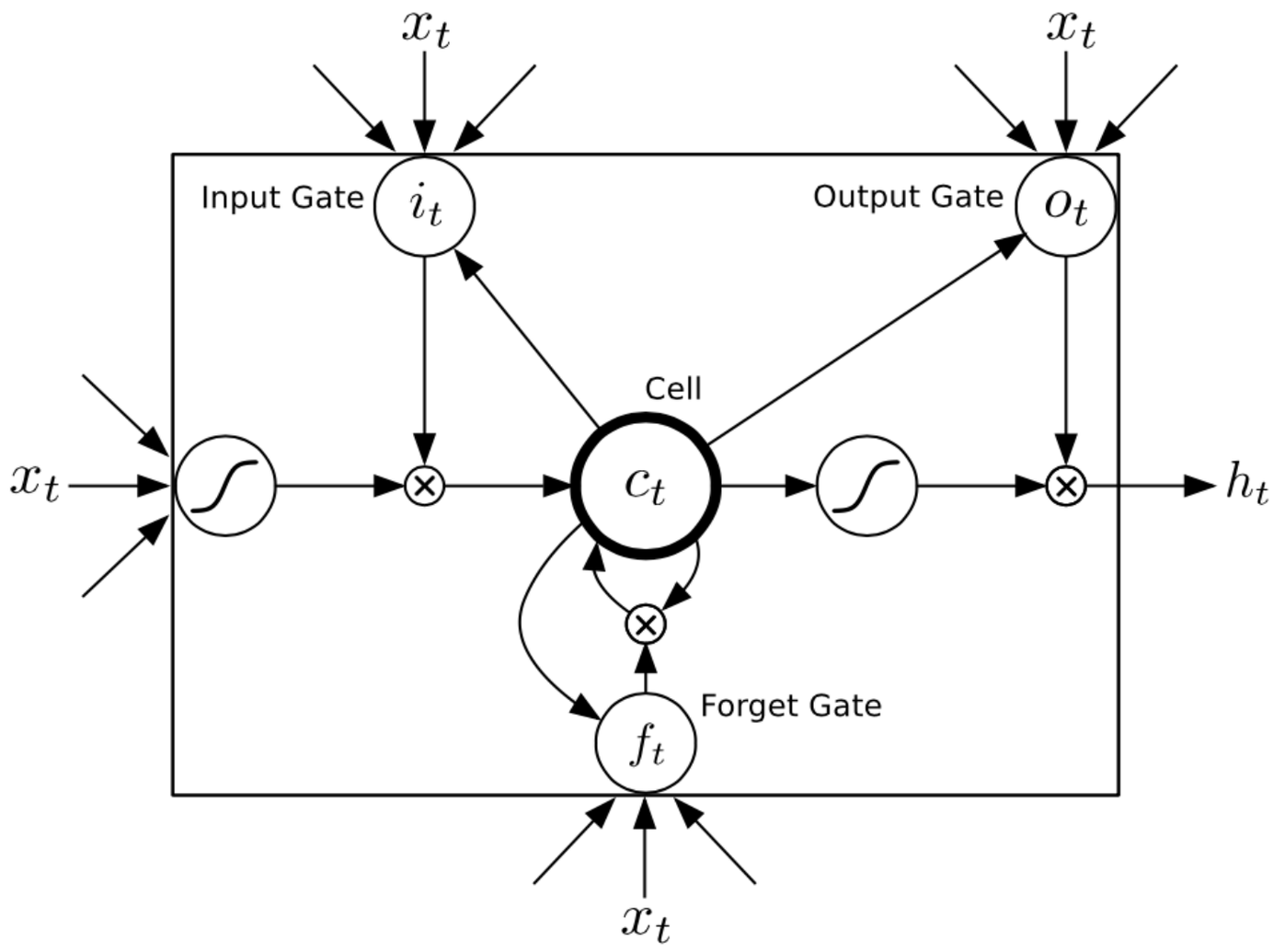


Source: http://colah.github.io/posts/2015-08-Understanding-LSTMs/

- Hard to track relationship between inputs & outputs
- “Vanishing” & “Exploding” gradients

LSTM

- Holds internal state (c_t) across timesteps
- *Gated* update of state based on input (x_t), previous output (h_{t-1}), and previous state (c_{t-1})



Generating Sequences With Recurrent Neural Networks (Graves 2012)

LSTM Equations

$$
\begin{aligned}
i_t &= \sigma\left(W_{xi}x_t + W_{hi}h_{t-1} + W_{ci}c_{t-1} + b_i\right) \\
f_t &= \sigma\left(W_{xf}x_t + W_{hf}h_{t-1} + W_{cf}c_{t-1} + b_f\right) \\
c_t &= f_t c_{t-1} + i_t \tanh\left(W_{xc}x_t + W_{hc}h_{t-1} + b_c\right) \\
o_t &= \sigma\left(W_{xo}x_t + W_{ho}h_{t-1} + W_{co}c_t + b_o\right) \\
h_t &= o_t \tanh(c_t)
\end{aligned}
$$

Generating Sequences With Recurrent Neural Networks (Graves 2012)

Example: Writing Fairy Tales

Source Data

- Grimm's Fairy Tales from Project Gutenberg
- ~500k characters
- 69 distinct characters

  "A certain king had a beautiful garden, and in the garden stood a tree which bore golden apples. These apples were always counted, and about the time when they began to grow ripe it was found that every night one of them was gone. The king became very angry at this, and ordered the gardener to keep watch all night under the tree."

Character-Level Language Modeling

- *Task:* Predict the next character from the previous k characters.
- *Input*: Sequence of binary vectors representing characters.
- *Output:* Probability distribution over characters.
- *Loss Function:* Probabilities of all wrong characters.

The notion "probability of a sentence" is an entirely useless one, under any known interpretation of this term. (Chomsky, 1969)

Keras implementation

```
model = Sequential()
for i, layer_size in enumerate(layer_sizes):
    last_lstm_layer = (i + 1 == len(layer_sizes))
    # last layer needs to only return its last activation
    kwargs = {"return_sequences": not last_lstm_layer}
    # need to tell Keras the input shape for first layer
    if i == 0:
        kwargs["input_shape"] = (maxlen, n_chars)
    model.add(LSTM(layer_size, **kwargs))
    model.add(Dropout(dropout_prob))
model.add(Dense(n_chars))
model.add(Activation('softmax'))
optimizer = RMSprop(lr=learning_rate, clipnorm=max_gradient_norm)
model.compile(loss='categorical_crossentropy', optimizer=optimizer)
```

Lasagne implementation

```
character_input_layer = InputLayer(shape=(batch_size, maxlen, n_chars))
layers = [character_input_layer]
for layer_size in layer_sizes:
    lstm_layer = LSTMLayer(
        incoming=layers[-1],
        num_units=layer_size,
        nonlinearity=tanh,
        hid_init=Glorot(Uniform),
        forgetgate=Gate(b=Constant(1.)))
    layers.append(lstm_layer)
    if dropout_prob > 0:
        dropout_layer = DropoutLayer(lstm_layer, p=dropout_prob)
        layers.append(dropout_layer)
# each LSTM layer returns a sequence of outputs, and we only need
# the last output to predict the next character, so we're going to use
# a SliceLayer to extract that last output vector
slice_layer = SliceLayer(layers[-1], indices=-1, axis=1)
# output is a Softmax distribution over charaters
output_layer = DenseLayer(
    slice_layer,
    num_units=n_chars,
    nonlinearity=softmax)
# we get a predicted vector of probabilities for each timestep,
# use deterministic=False to activate Dropout.
training_output = get_output(output_layer, deterministic=False)

# Slicing through the output vector since we only want the
# probabilities coming from the last timestep
# training_output = training_output[:, -1]

# create a symbolic variable for (minibatch, n_chars) binary
# encoding of desired outputs
target_probabilities = T.matrix('target_probabilities')

# train network to minimize cross-entropy
cost = categorical_crossentropy(target_probabilities, training_output).mean()

# retrieve all parameters from the network
all_params = get_all_params(output_layer, trainable=True)
gradients = theano.grad(cost, all_params)
```

```
gradients = [
    norm_constraint(
        g,
        MAX_GRADIENT_NORM,
        # take norm along all axes except for the 0'th, which corresponds
        # to minibatches
        norm_axes=tuple(range(1, g.ndim + 1)))
    for g in gradients
]
# using RMSprop to determine updates for each parameter
updates = rmsprop(gradients, all_params, learning_rate=LEARNING_RATE)

# Theano functions for training and computing cost
train_fn = theano.function(
    [character_input_layer.input_var, target_probabilities],
    cost,
    updates=updates)

cost_fn = theano.function(
    [character_input_layer.input_var, target_probabilities],
    # take the mean since the cost is otherwise computed per
    # sample
    cost)

# The crucial difference here is that we do a deterministic forward pass
# through the network, disabling Dropout layers.
deterministic_output = get_output(output_layer, deterministic=True)
predict_fn = theano.function(
    [character_input_layer.input_var],
    deterministic_output)
```

Chainer implementation

```
# first create weights and model layers, the actual semantics of
# how we use these weights will be defined later
model = chainer.FunctionSet(
    l1_x=F.Linear(n_chars, 4 * hidden1_size),
    l1_h=F.Linear(hidden1_size, 4 * hidden1_size),
    l2_x=F.Linear(hidden1_size, 4 * hidden2_size),
    l2_h=F.Linear(hidden1_size, 4 * hidden2_size),
    output=F.Linear(hidden2_size, n_chars)
)
for param in model.parameters:
    param[:] = np.random.uniform(-0.1, 0.1, param.shape)

if cuda.available:
    cuda.get_device(0).use()
    model.to_gpu()

# now that we've transferred the model weights to the GPU,
# set  up an optimizer that we're going to use to train the model
optimizer = RMSprop(lr=LEARNING_RATE)
optimizer.setup(model)
```

```
# the actual input-to-output mapping is defined in Chainer
# via its "Define-By-Run" model of just passing intermediate
# values through successive transformations
def forward_one_step(X_batch, state, train=True):
    x = chainer.Variable(X_batch, volatile=not train)
    old_hidden1 = state["hidden1"]
    x_to_h = model.l1_x(x)
    h_to_h = model.l1_h(old_hidden1)
    hidden1_input = x_to_h + h_to_h

    old_cell1 = state["cell1"]
    cell1, hidden1 = F.lstm(old_cell1, hidden1_input)
    hidden1_dropout = F.dropout(hidden1, dropout_prob, train=train)

    hidden2_old = state["hidden2"]
    hidden2_input = model.l2_x(hidden1_dropout) + model.l2_h(hidden2_old)
    cell2_old = state["cell2"]
    cell2, hidden2 = F.lstm(cell2_old, hidden2_input)
    hidden2_dropout = F.dropout(hidden2, dropout_prob, train=train)
    logits = model.output(hidden2_dropout)
    probs = F.softmax(logits)
    state = {
        'cell1': cell1,
        'hidden1': hidden1,
        'cell2': cell2,
        'hidden2': hidden2,
    }
    return state, probs
```

```
def predict_batch(X_batch, train=True):
    """
    In imitation of the stateless LSTM we made with Keras
    ignoring state propagation across forward calls and instead
    only allowing the contents of each short sequence sample to
    affect the model's output.
    """
    batch_size, maxlen = X_batch.shape[:2]
    state = make_initial_state(batch_size=batch_size, train=train)
    for j in range(maxlen):
        state, probs = forward_one_step(
            X_batch[:, j, :],
            state,
            train=train)
    return probs

def cost_fn(X, y, train=False):
    accum_loss = chainer.Variable(xp.zeros((), dtype=np.float32))
    for i in range(0, len(y), batch_size):
        X_batch = X[i:i + batch_size]
        y_batch = y[i:i + batch_size]
        probs = predict_batch(X_batch, train=False)
        y_batch_var = chainer.Variable(y_batch, volatile=True)
        loss = F.softmax_cross_entropy(probs, y_batch_var)
        accum_loss += loss
    return cuda.to_cpu(accum_loss) / len(y)
```

```
def train_fn(X, y, batch_size=BATCH_SIZE, shuffle=True):
    if shuffle:
        indices = np.arange(len(y))
        np.random.shuffle(indices)
        X, y = X[indices], y[indices]
    optimizer.zero_grads()
    for i in range(0, len(y), batch_size):
        X_batch = X[i:i + batch_size, :]
        y_batch = y[i:i + batch_size]
        probs = predict_batch(X_batch, train=True)
        y_batch_var = chainer.Variable(y_batch, volatile=False)
        loss = F.softmax_cross_entropy(probs, y_batch_var)
        # compute gradients
        loss.backward()
        # rescale gradients to not exceed norm threshold
        optimizer.clip_grads(MAX_GRADIENT_NORM)
        # apply RMSpop weight update to weights in the function set
        optimizer.update()
```

Output, epoch #1

“'I the kas would ontile were her werl now I heast of the sonenund lest enct her ouk that pistered the with of fean, wile that the fing wared me in the parled to the bees if the sther gound.' Then whan shund again, the seeps of the wame went on the coot; be he as deated sime out of the the, and boked yither”

Output, epoch #2

“The bettle seet resent throw in his sell and seard ney woor and see betore.' 'qoat rawed, with one lyor wand her, he lettle she sauded out of one to the shore sanded off the would be wonderth put her that she once sen which neved becound, and the toot and saw a loodser, and said her one will be soon arl dead.' But I will go and kered to the bear reppined.'”

Output, epoch #10

"So the old woman was clacked upon his head, and mise could not run off and berought to courterus, for the fisherman was that the young grey back, and the cat was before to the church, and the grandmasters were off back."

Output, epoch #15

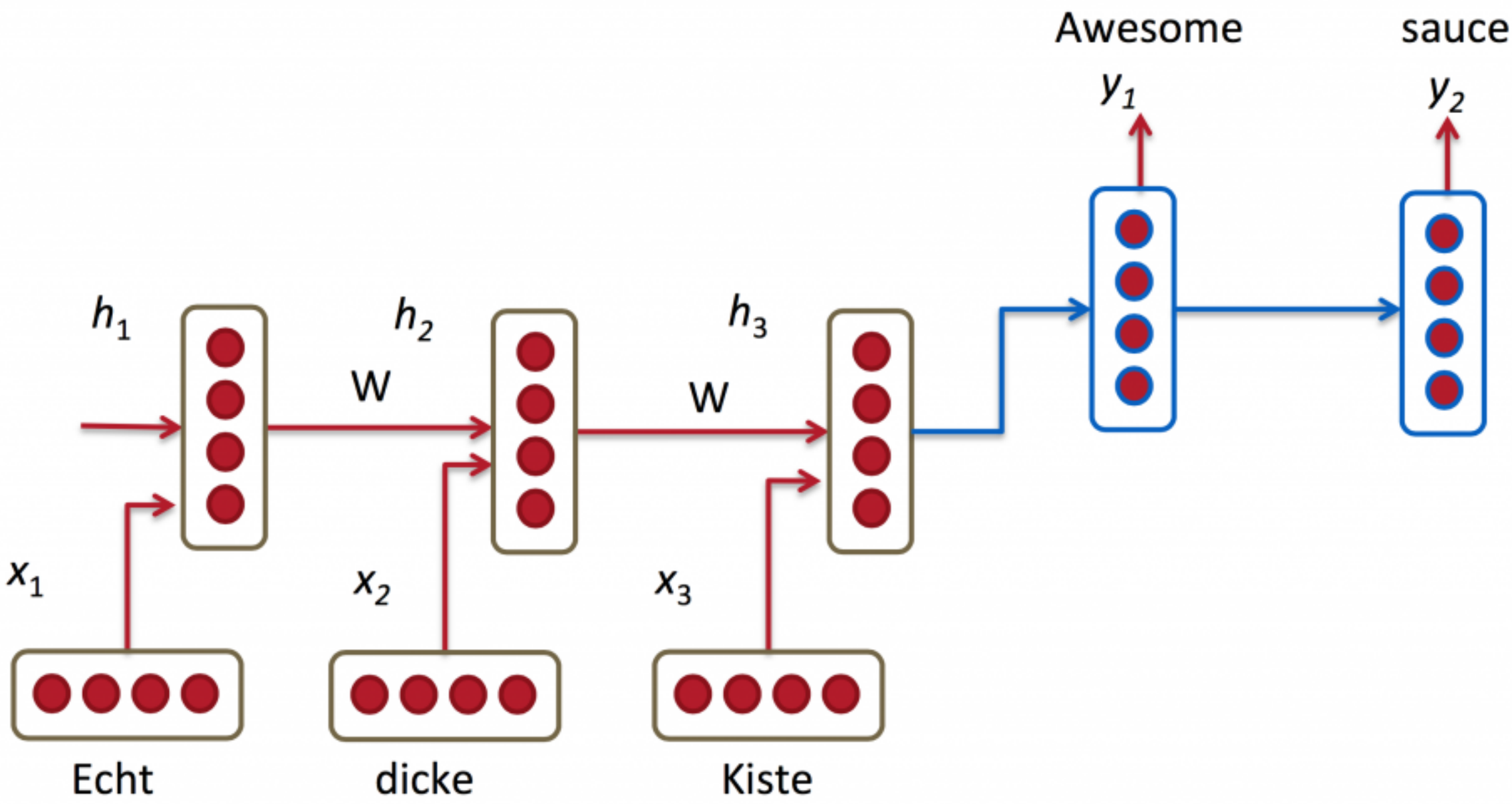
“The king came and said, 'Give me a moon.' 'Ah,' said the wolf, 'the king was so well to go out to him her way in the sea, for a wood would have nothing to wait all yourself. 'That was bring the church of the chamber,' said the old woman, 'and take some his horse.'”

Output, epoch #40

“The bear was standing by the side of the stream, and said to him, 'What a way into the forest the golden house if you have been the world when they shall have sent me forth, and I will soon find you down in an earth.' The man was going on the house, and sent the fire, and she said, 'If you have a fat good companion for the princess.'”

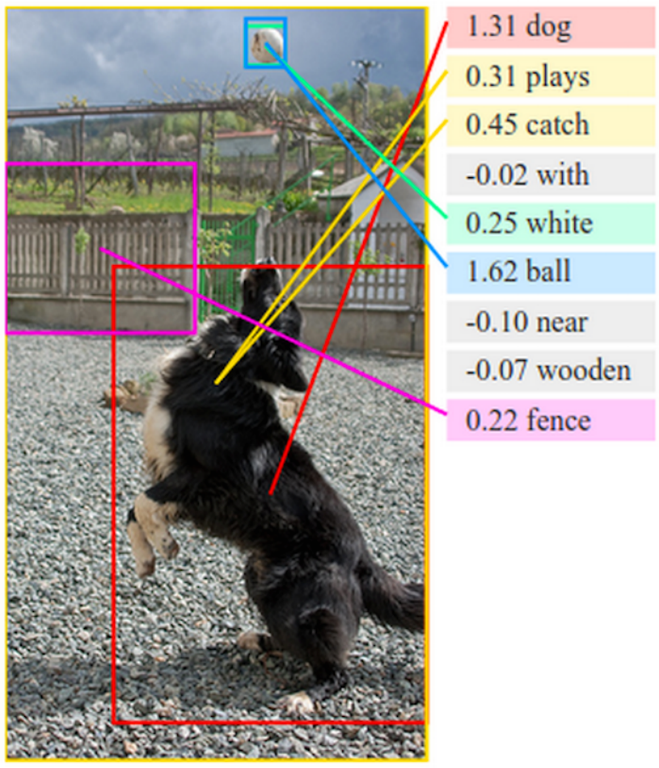
Other RNN Tasks

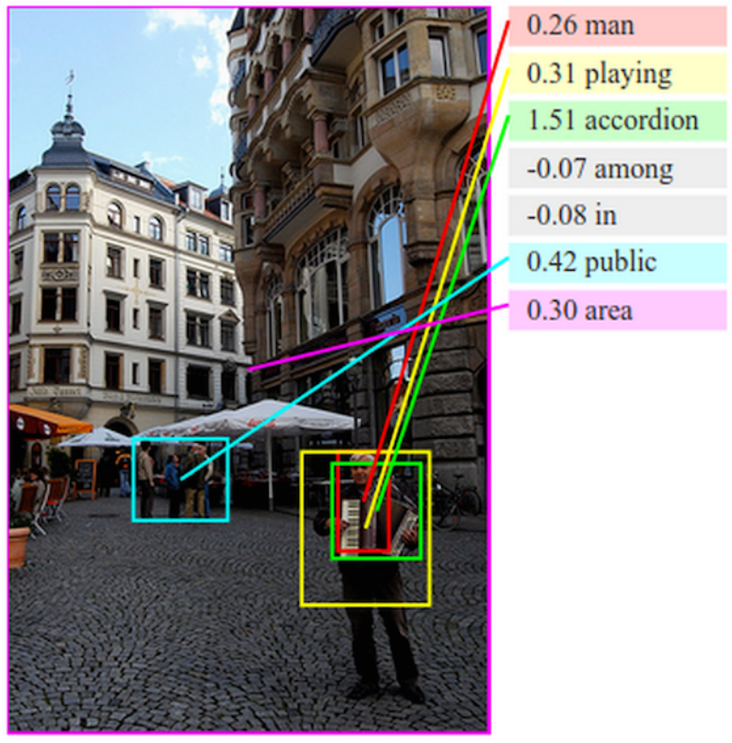
Machine Translation

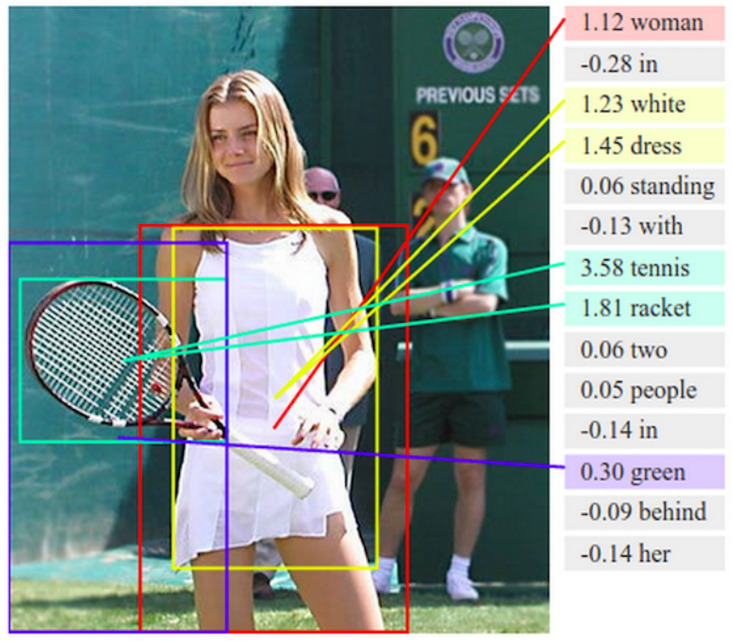


Source: http://cs224d.stanford.edu/lectures/CS224d-Lecture8.pdf

Captioning







Source: http://cs.stanford.edu/people/karpathy/deepimagesent/

Computational Biology

- Predicting mRNA splicing from DNA sequence
- Predicting at which residues a protein will be cut by the proteasome (or other proteases)
- **Predicting binding of long peptides to Class II MHC molecules for immune recognition by Helper T-cells.**

Thanks!