Deep Learning for Intelligent Video Analysis

Tao Mei, Senior Researcher
Cha Zhang, Principal Applied Science Manager
Microsoft AI & Research

Microsoft

Agenda

- Introduction for Cognitive Toolkit (Cha)
  - CNTK overview
  - CNTK for image/video tasks
- Break
- Intelligent video analysis (Tao)

Microsoft Cognitive Toolkit (CNTK)

- Microsoft's open-source deep-learning toolkit
  - https://github.com/Microsoft/CNTK
  - Created by Microsoft Speech researchers in 2012, "Computational Network Toolkit"
  - Open sourced on CodePlex in Apr 2015
  - On GitHub since Jan 2016 under MIT license, and renamed to "Cognitive Toolkit"
  - 12,700+ GitHub stars
    - Third among all DL toolkits
  - 150 contributors from MIT, Stanford, Nvidia, Intel and many others
  - Internal == External

Microsoft Cognitive Toolkit

- Runs over 80% Microsoft internal DL workload
- 1st-class on Linux and Windows, docker support
- Rich API support
  - Mostly implemented in C++ (train and eval)
  - Low level + high level Python API
  - R and C# API for train and eval
  - Universal Windows Platform (UWP), Java and Spark support
  - Built-in readers for distributed learning
  - Keras backend support (Beta)
  - Model compression (Fast binarized evaluation)
- Latest version: v2.2 (Sep. 15, 2017)

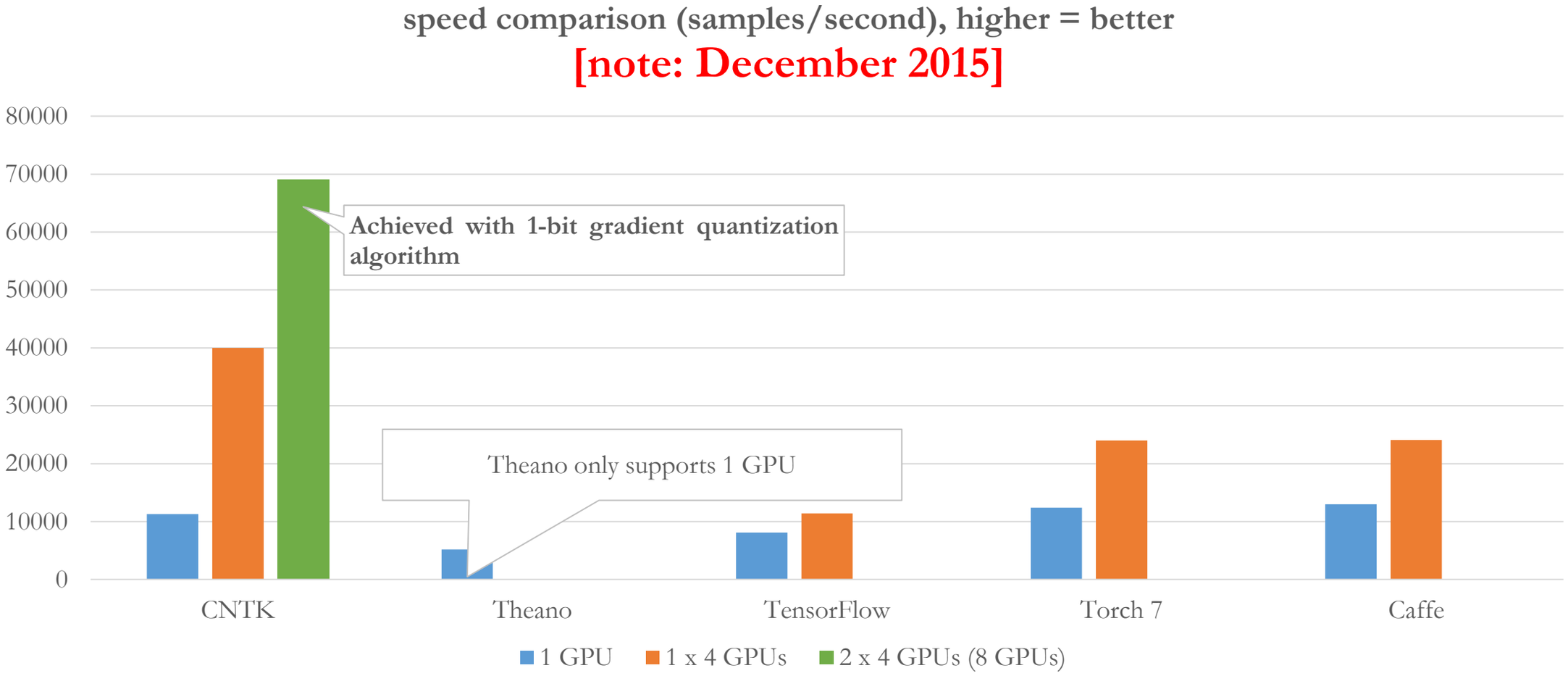
Toolkit Benchmark

http://dlbench.comp.hkbu.edu.hk/
Benchmarking by HKBU, Version 8
Single Tesla K80 GPU, CUDA: 8.0 CUDNN: v5.1

Caffe: 1.0rc5(39f28e4)
CNTK: 2.0 Beta10(1ae666d)
MXNet: 0.93(32dc3a2)
TensorFlow: 1.0(4ac9c09)
Torch: 7(748f5e3)

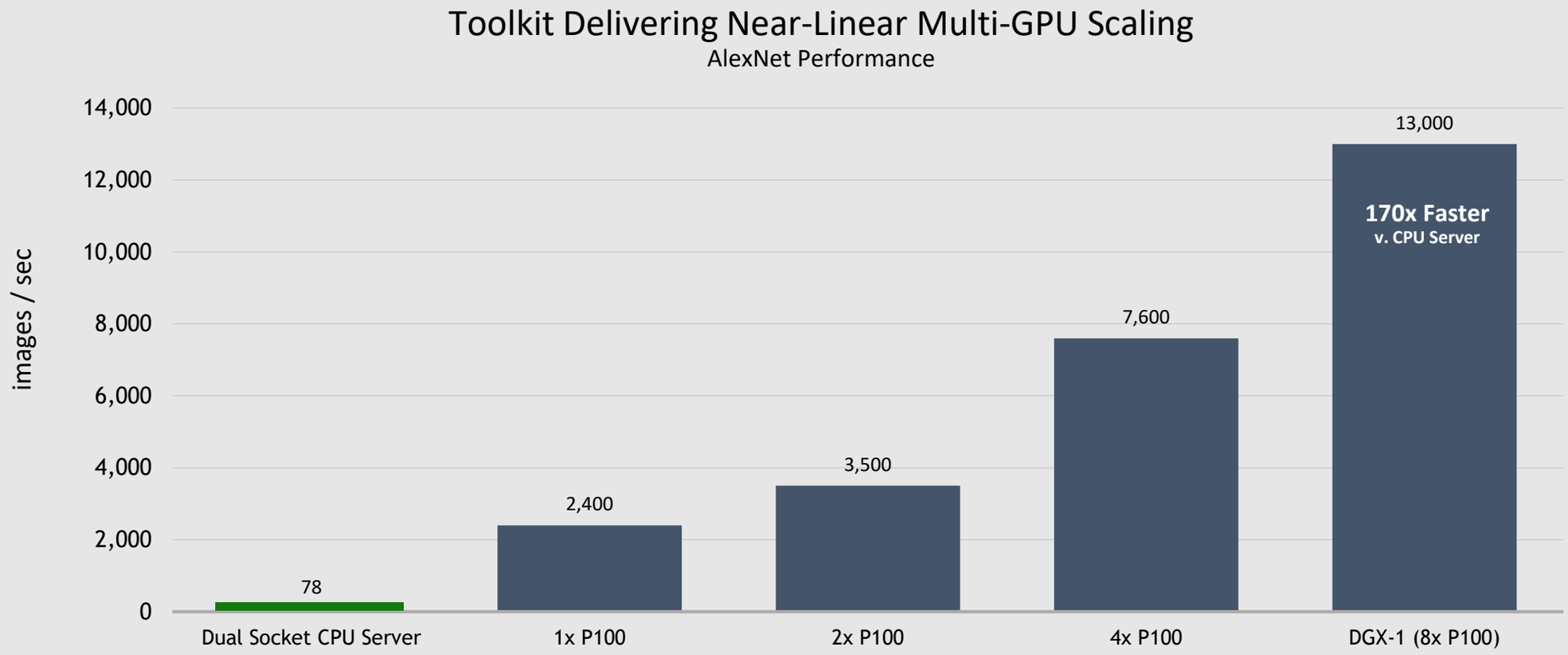
	Caffe	Cognitive Toolkit	MxNet	TensorFlow	Torch
FCN5 (1024)	55.329ms	**51.038ms**	60.448ms	62.044ms	52.154ms
AlexNet (256)	36.815ms	**27.215ms**	28.994ms	103.960ms	37.462ms
ResNet (32)	143.987ms	**81.470ms**	84.545ms	181.404ms	90.935ms
LSTM (256) (v7 benchmark)	-	**43.581ms** **(44.917ms)**	288.142ms (284.898ms)	- (223.547ms)	1130.606ms (906.958ms)

“CNTK is production-ready: State-of-the-art accuracy, efficient, and scales to multi-GPU/multi-server.”



MICROSOFT COGNITIVE TOOLKIT

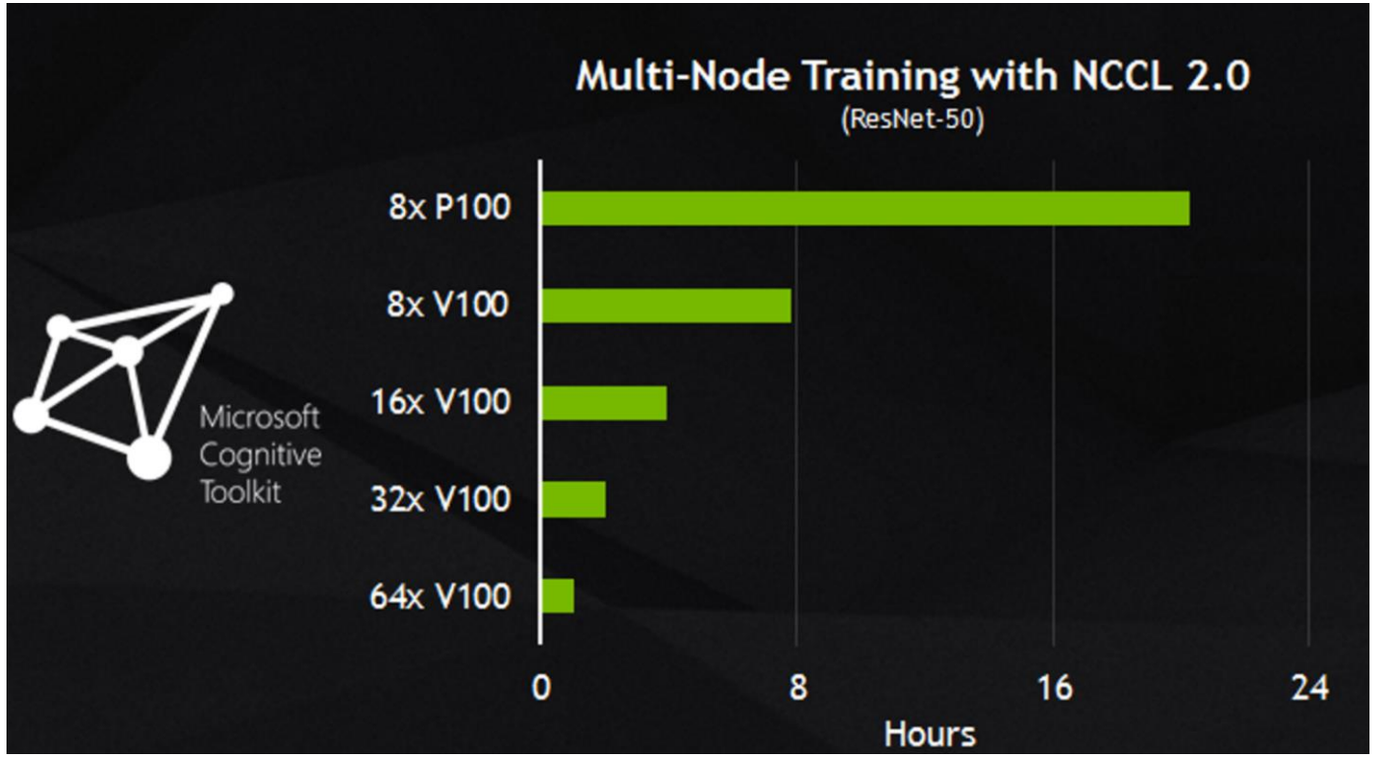
First Deep Learning Framework Fully Optimized for Pascal



AlexNet training batch size 128, Grad Bit = 32, Dual socket E5-2699v4 CPUs (total 44 cores)
CNTK 2.0b3 (to be released) includes cuDNN 5.1.8, NCCL 1.6.1, NVLink enabled

Microsoft

Scale with NCCL 2.0



GTC, May 2017
NCCL 2 support in CNTK v2.2

nvidianews.nvidia.com/news/nvidia-and-microsoft-accelerate-ai-together

NVIDIA

DRIVERS | PRODUCTS | DEEP LEARNING AND AI | COMMUNITIES | SUPPORT | SHOP

NEWSROOM | Multimedia | Executive Bios | Media Contacts | In

News

NVIDIA and Microsoft Accelerate AI Together

Monday, November 14, 2016

GPU-Accelerated Microsoft Cognitive Toolkit Now Available in the Cloud on Microsoft Azure and On-Premises with NVIDIA DGX-1

SC16 -- To help companies join the AI revolution, NVIDIA today announced a collaboration with Microsoft to accelerate AI in the enterprise.

Using the first purpose-built enterprise AI framework optimized to run on NVIDIA® Tesla® GPUs in Microsoft Azure or on-premises, enterprises now have an AI platform that spans from their data center to Microsoft's cloud.

"Every industry has awoken to the potential of AI," said Jen-Hsun Huang, founder and chief executive officer, NVIDIA. "We've worked with Microsoft to create a lightning-fast AI platform that is available from on-premises with our DGX-1™ supercomputer to the Microsoft Azure cloud. With Microsoft's global reach, every company around the world can now tap the power of AI to transform their business."

"We're working hard to empower every organization with AI, so that they can make smarter products and solve some of the world's most pressing problems," said Harry Shum, executive vice president of the Artificial Intelligence and Research Group at Microsoft. "By working closely with NVIDIA and harnessing the power of GPU-accelerated systems, we've made Cognitive Toolkit and Microsoft Azure the fastest, most versatile AI platform. AI is now within reach of any business."

This jointly optimized platform runs the new Microsoft Cognitive Toolkit (formerly CNTK) on NVIDIA GPUs, including the NVIDIA DGX-1™ supercomputer, which uses Pascal™ architecture GPUs with NVLink™ interconnect technology, and on Azure N-Series virtual machines, currently in preview. This combination delivers unprecedented performance and ease of use when using data for deep learning.

As a result, companies can harness AI to make better decisions, offer new products and services faster and provide better customer experiences. This is causing every industry to implement AI. In just two years, the number of companies NVIDIA collaborates with on deep learning has jumped 194x to over 19,000. Industries such as healthcare, life sciences, energy, financial services, automotive and manufacturing are benefiting from deeper insight on extreme amounts of data.

Scale to 1,000 GPUs

Image: Cray

Microsoft, Cray claim deep learning breakthrough on supercomputers

Steve Ranger
ZDNet

A team of researchers from Microsoft, Cray, and the Swiss National Supercomputing Centre (CSCS) have been working on a project to speed up the use of deep learning algorithms on supercomputers.

The team have scaled the Microsoft Cognitive Toolkit -- an open-source suite that trains deep learning algorithms -- to more than 1,000 Nvidia Tesla P100 GPU accelerators on the Swiss centre's Cray XC50 supercomputer, which is nicknamed Piz Daint.

Microsoft

What is CNTK?

- CNTK expresses (nearly) **arbitrary neural networks** by composing simple building blocks into complex **computational networks**, supporting relevant network types and applications.

What is CNTK?

Example: 2-hidden layer feed-forward NN

$$h_1 = \sigma(W_1 x + b_1)$$
$$h_2 = \sigma(W_2 h_1 + b_2)$$
$$P = \mathrm{softmax}(W_{\mathrm{out}} h_2 + b_{\mathrm{out}})$$

with input $x \in \mathrm{R}^M$

What is CNTK?

Example: 2-hidden layer feed-forward NN

$$h_1 = \sigma(W_1 x + b_1)$$
$$h_2 = \sigma(W_2 h_1 + b_2)$$
$$P = \text{softmax}(W_{\text{out}} h_2 + b_{\text{out}})$$

with input $x \in \mathrm{R}^M$ and one-hot label $y \in \mathrm{R}^J$
and cross-entropy training criterion

$$ce = y^{\mathrm{T}} \log P$$
$$\sum_{\text{corpus}} ce = \max$$

What is CNTK?

example: 2-hidden layer feed-forward NN

$$h_1 = \sigma(W_1 x + b_1)$$
$$h_2 = \sigma(W_2 h_1 + b_2)$$
$$P = \text{softmax}(W_{\text{out}} h_2 + b_{\text{out}})$$

```
h1 = sigmoid (x  @ w1   + b1)
h2 = sigmoid (h1 @ w2   + b2)
P  = softmax (h2 @ wout + bout)
```

with input $x \in \mathrm{R}^M$ and one-hot label $y \in \mathrm{R}^J$
and cross-entropy training criterion

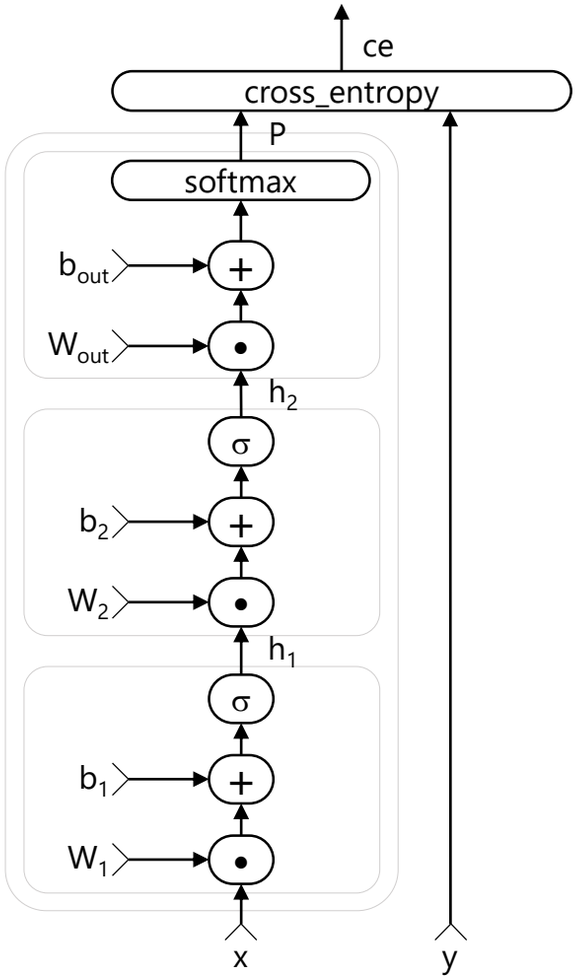
$$ce = y^{\mathrm{T}} \log P$$
$$\sum_{\text{corpus}} ce = \max$$

```
ce = cross_entropy (P, y)
```

What is CNTK?

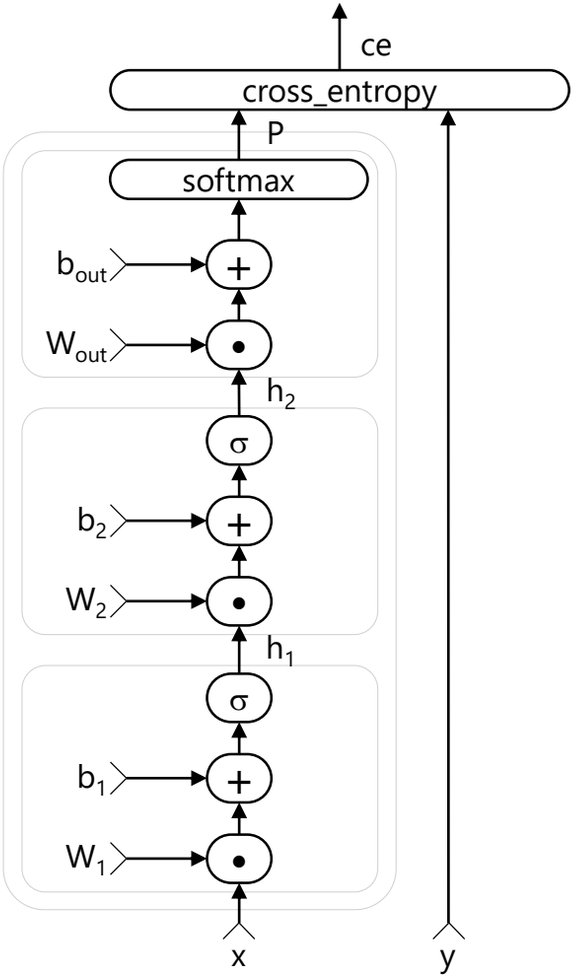
```
h1 = sigmoid (x  @ w1   + b1)
h2 = sigmoid (h1 @ w2   + b2)
P  = softmax (h2 @ wout + bout)
ce = cross_entropy (P, y)
```

What is CNTK?




```
h1 = sigmoid (x  @ w1   + b1)
h2 = sigmoid (h1 @ w2   + b2)
P  = softmax (h2 @ wout + bout)
ce = cross_entropy (P, y)
```

What is CNTK?



- Nodes: functions (primitives)
  - Can be composed into reusable composites
- Edges: values
  - Incl. tensors, sparse
- Automatic differentiation
  - $\partial \mathcal{F} / \partial \text{in} = \partial \mathcal{F} / \partial \text{out} \cdot \partial \text{out} / \partial \text{in}$
- Deferred computation → execution engine
- Editable, clonable

LEGO-like composability allows CNTK to support wide range of networks & applications

Symbolic Loops over Sequential Data

Extend our example to a recurrent network (RNN)

$$
\begin{aligned}
h_1 &= \sigma(\mathbf{W}_1\, x + b_1) \\
h_2 &= \sigma(\mathbf{W}_2\, h_1 + b_2) \\
P &= \mathrm{softmax}(\mathbf{W}_{\mathrm{out}}\, h_2 + b_{\mathrm{out}}) \\
ce &= L^{\mathrm{T}} \log P \\
\textstyle\sum_{\mathrm{corpus}} ce &= \max
\end{aligned}
$$

Symbolic Loops over Sequential Data

Extend our example to a recurrent network (RNN)

$$h_1(t) = \sigma(\mathbf{W}_1\, x(t) + \mathbf{R}_1\, h_1(t\text{-}1) + b_1)$$

```
h1 = sigmoid(x  @ W1 + past_value(h1) @ R1 + b1)
```

$$h_2(t) = \sigma(\mathbf{W}_2\, h_1(t) + \mathbf{R}_2\, h_2(t\text{-}1) + b_2)$$

```
h2 = sigmoid(h1 @ W2 + past_value(h2) @ R2 + b2)
```

$$P(t) = \text{softmax}(\mathbf{W}_{\text{out}}\, h_2(t) + b_{\text{out}})$$

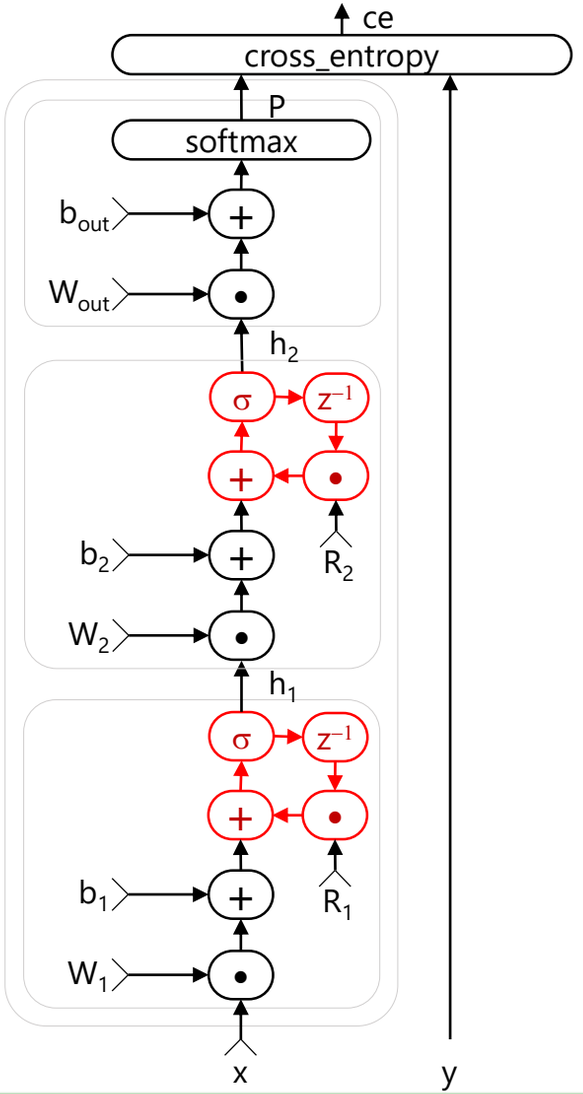
```
P  = softmax(h2 @ Wout + bout)
```

$$ce(t) = L^{\text{T}}(t) \log P(t)$$

```
ce = cross_entropy(P, L)
```

$$\sum_{\text{corpus}} ce(t) = \max$$

Symbolic Loops over Sequential Data

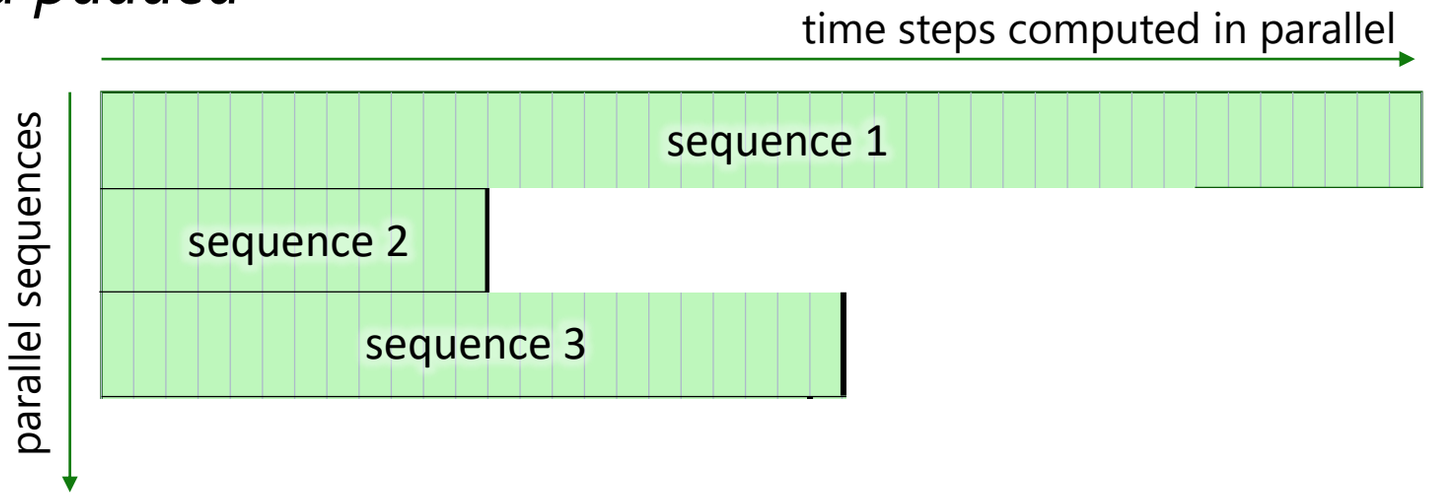



```
h1 = sigmoid(x  @ W1 + past_value(h1) @ R1 + b1)
h2 = sigmoid(h1 @ W2 + past_value(h2) @ R2 + b2)
P  = softmax(h2 @ Wout + bout)
ce = cross_entropy(P, L)
```

- CNTK automatically unrolls cycles at *execution time*
  - cycles are detected with Tarjan's algorithm
- Efficient and composable

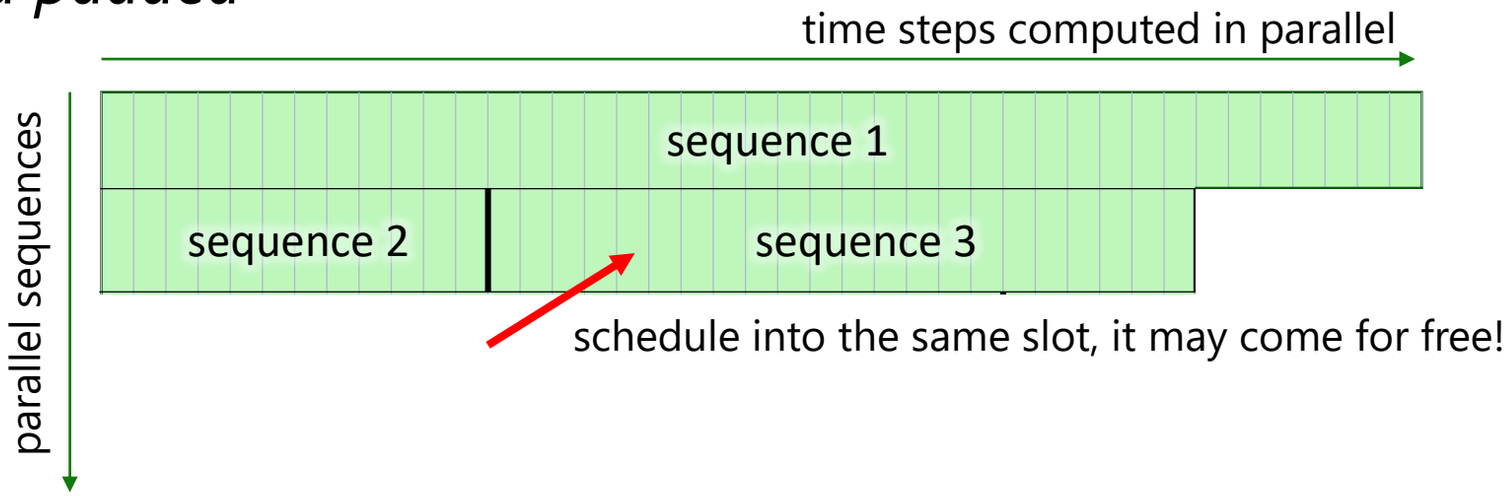
Batch-Scheduling of Variable-Length Sequences

- Minibatches containing sequences of different lengths are automatically packed *and padded*



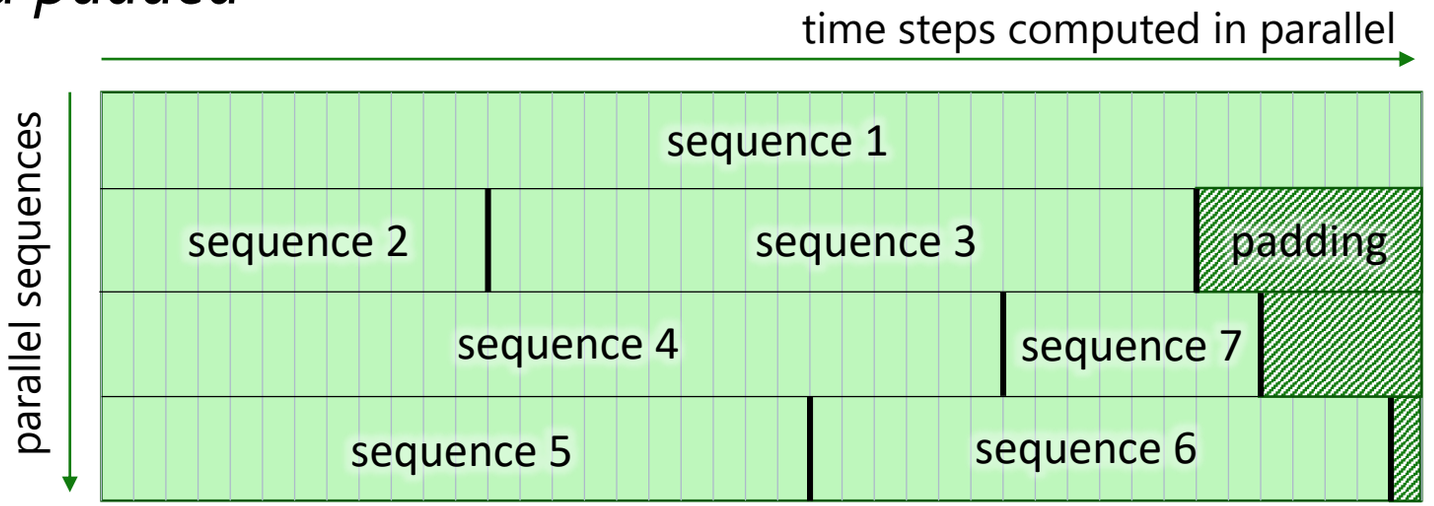
Batch-Scheduling of Variable-Length Sequences

- Minibatches containing sequences of different lengths are automatically packed *and padded*



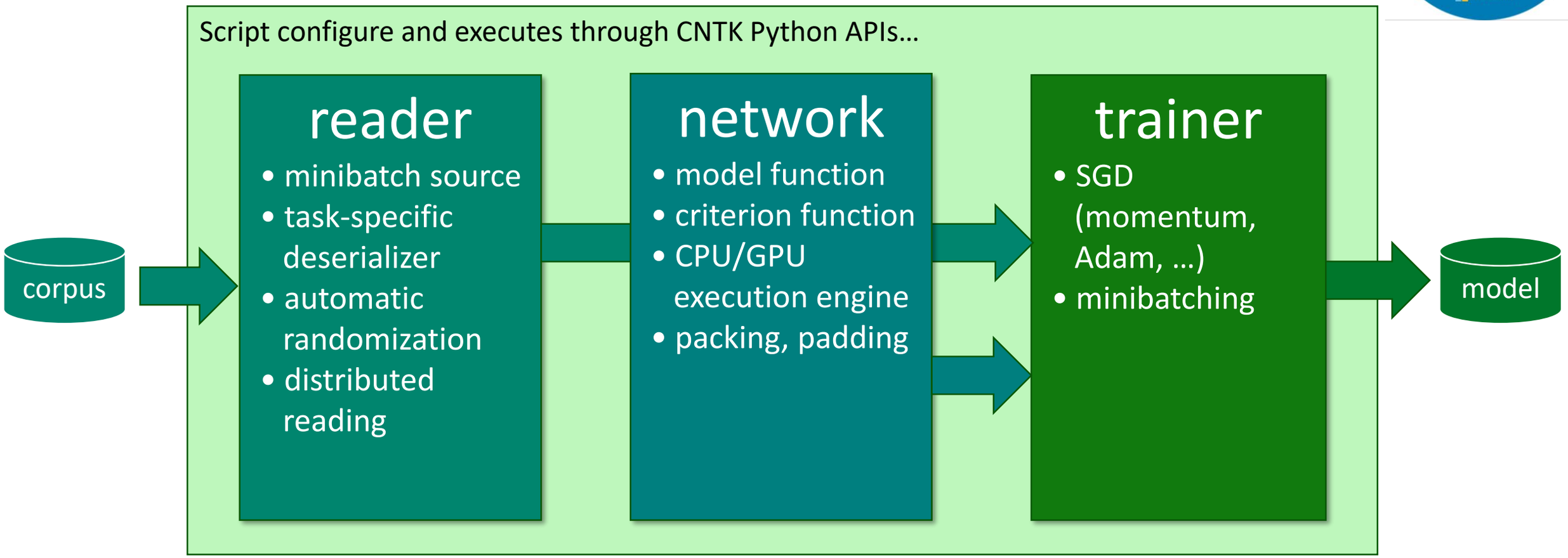
Batch-Scheduling of Variable-Length Sequences

- Minibatches containing sequences of different lengths are automatically packed *and padded*



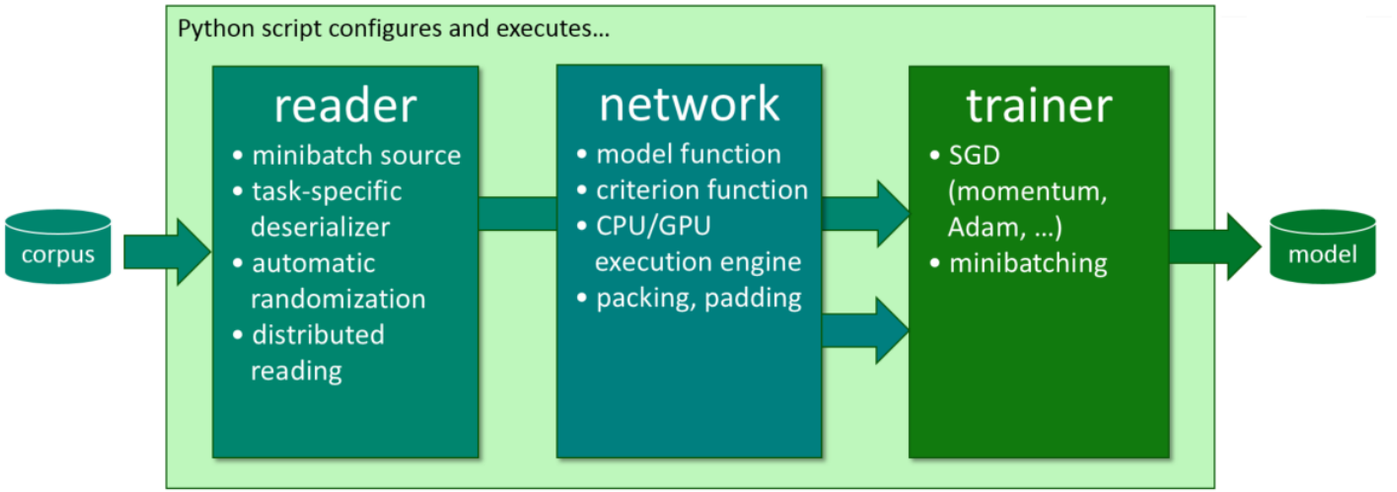
- Fully transparent batching
  - Recurrent → CNTK unrolls, handles sequence boundaries
  - Non-recurrent operations → parallel
  - Sequence reductions → mask

CNTK Workflow

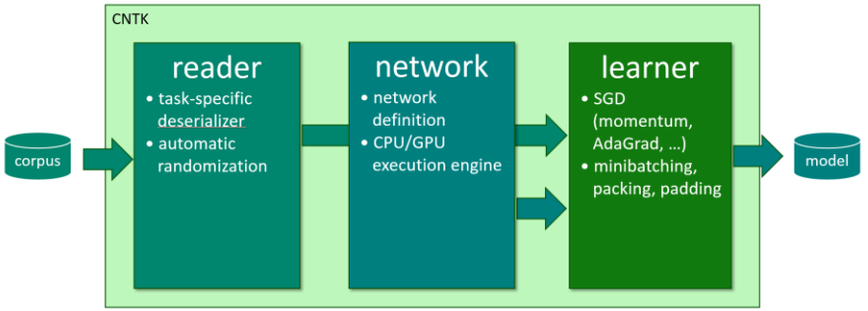


As Easy as 1-2-3

```
from cntk import *

# reader
def create_reader(path, is_training):
    ...

# network
def create_model_function():
    ...
def create_criterion_function(model):
    ...

# trainer (and evaluator)
def train(reader, model):
    ...
def evaluate(reader, model):
    ...

# main function
model = create_model_function()

reader = create_reader(..., is_training=True)
train(reader, model)

reader = create_reader(..., is_training=False)
evaluate(reader, model)
```



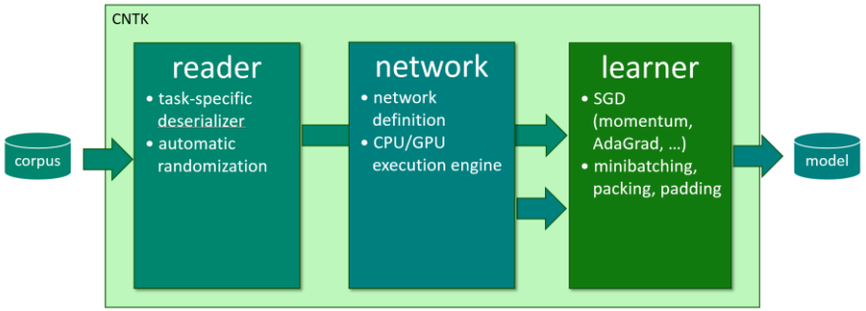
Workflow



- Prepare data
- Configure reader, network, learner (Python)
- Train:

```
python my_cntk_script.py
```

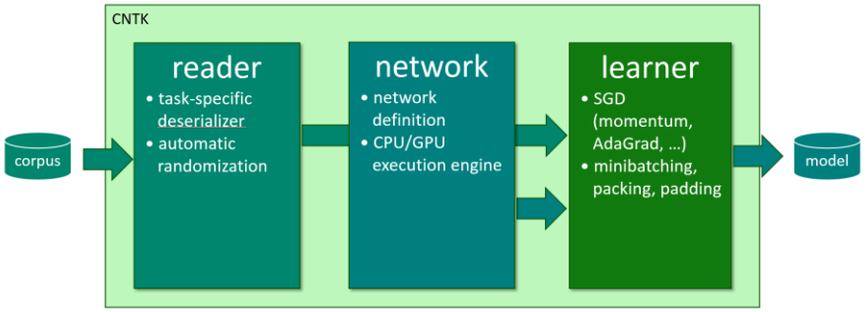
Prepare Data: Reader




```
def create_reader(map_file, mean_file, is_training):
```

```
    # deserializer
    return MinibatchSource(ImageDeserializer(map_file, StreamDefs(
        features = StreamDef(field='image', transforms=transforms), '
        labels   = StreamDef(field='label', shape=num_classes)
    )), randomize=is_training, epoch_size = INFINITELY_REPEAT if is_training else FULL_DATA_SWEEP)
```

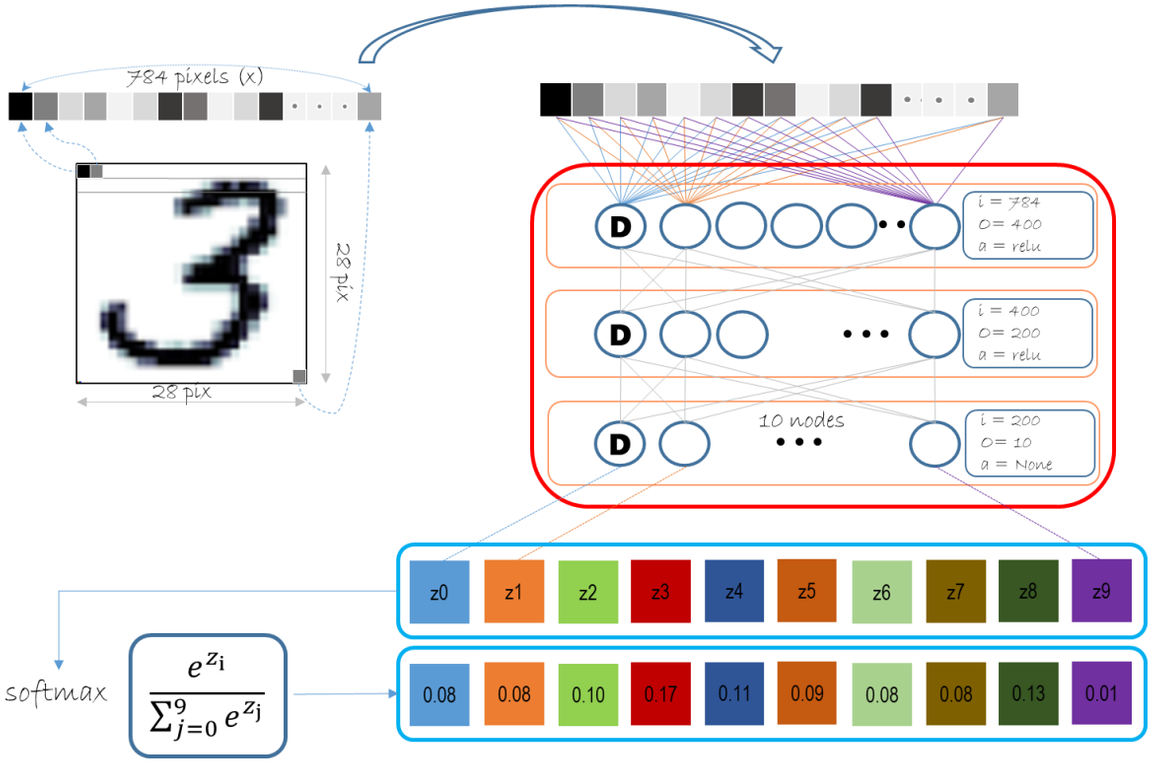
Prepare Data: Reader




```
def create_reader(map_file, mean_file, is_training):
    # image preprocessing pipeline
    transforms = [
        ImageDeserializer.crop(crop_type='Random', ratio=0.8, jitter_type='uniRatio')
        ImageDeserializer.scale(width=image_width, height=image_height, channels=num_channels,
                                interpolations='linear'),
        ImageDeserializer.mean(mean_file)
    ]
    # deserializer
    return MinibatchSource(ImageDeserializer(map_file, StreamDefs(
        features = StreamDef(field='image', transforms=transforms), '
        labels   = StreamDef(field='label', shape=num_classes)
    )), randomize=is_training, epoch_size = INFINITELY_REPEAT if is_training else FULL_DATA_SWEEP)
```

- Automatic on-the-fly randomization important for large data sets
- Readers compose, e.g. image → text caption

Prepare Network: Multi-Layer Perceptron



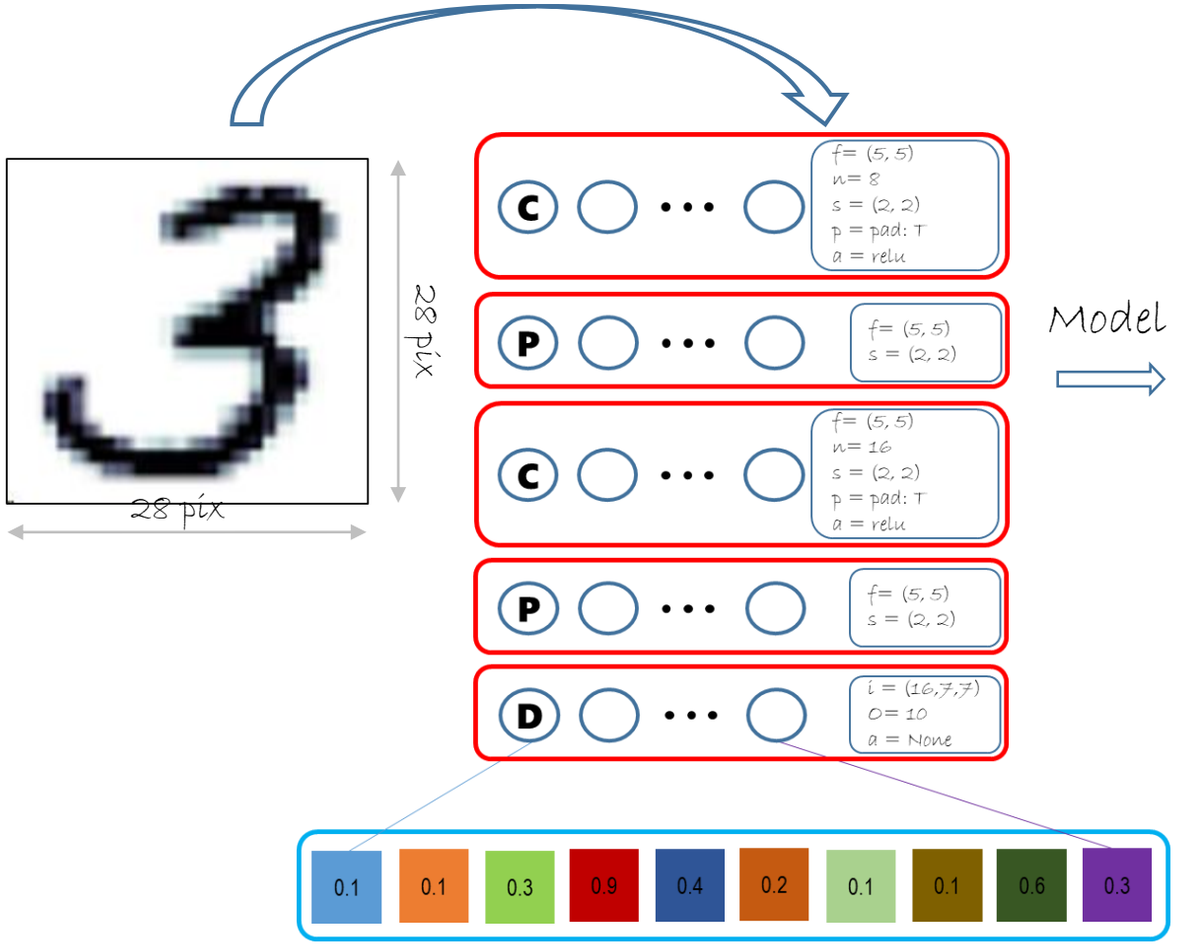
Model

```
z = model(x):
        h1 = Dense(400, act = relu)(x)
        h2 = Dense(200, act = relu)(h1)
        r  = Dense(10,  act = None)(h2)
        return r
```

Loss

```
cross_entropy_with_softmax(z,Y)
```

Prepare Network: Convolutional Neural Network

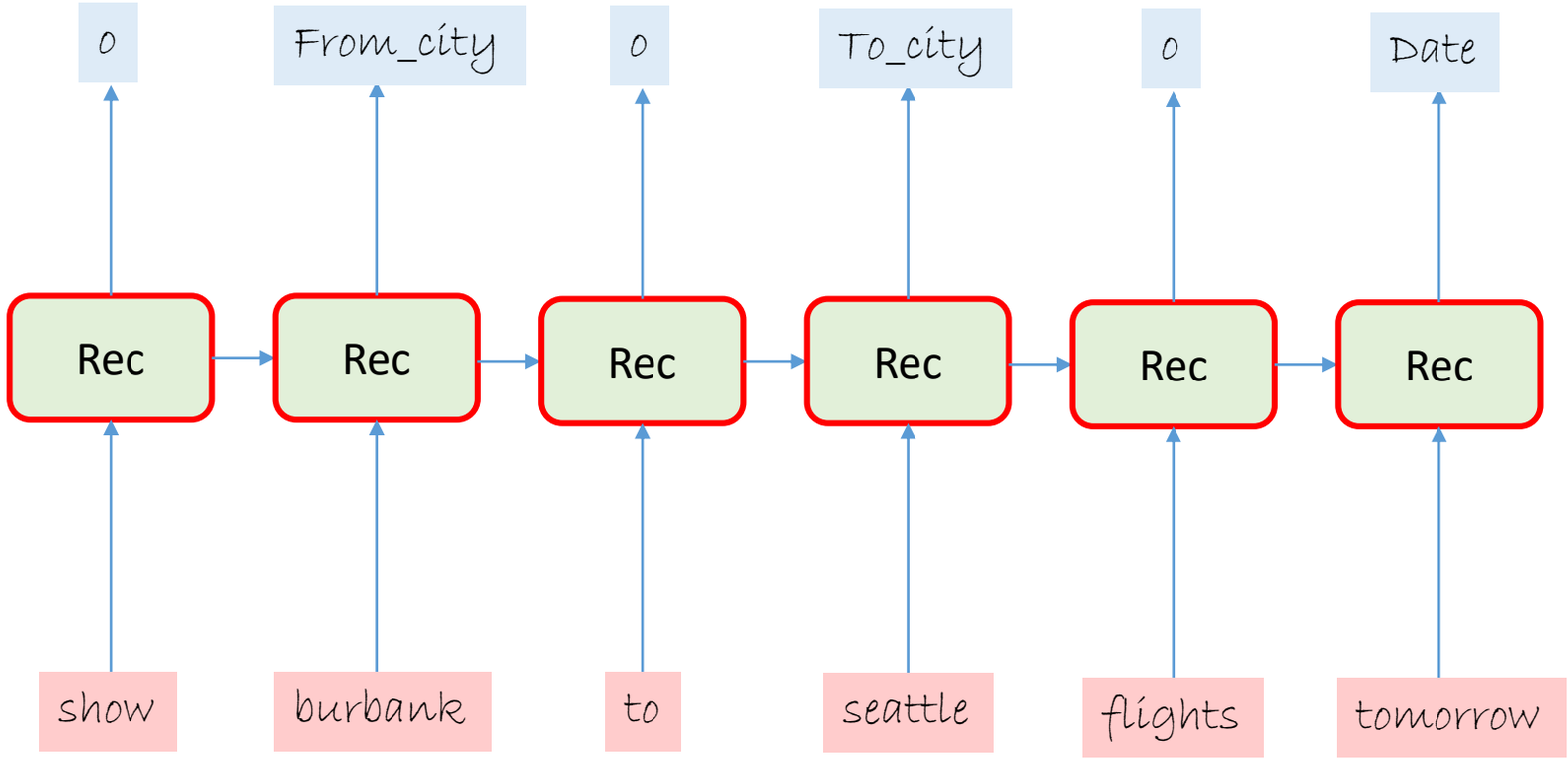


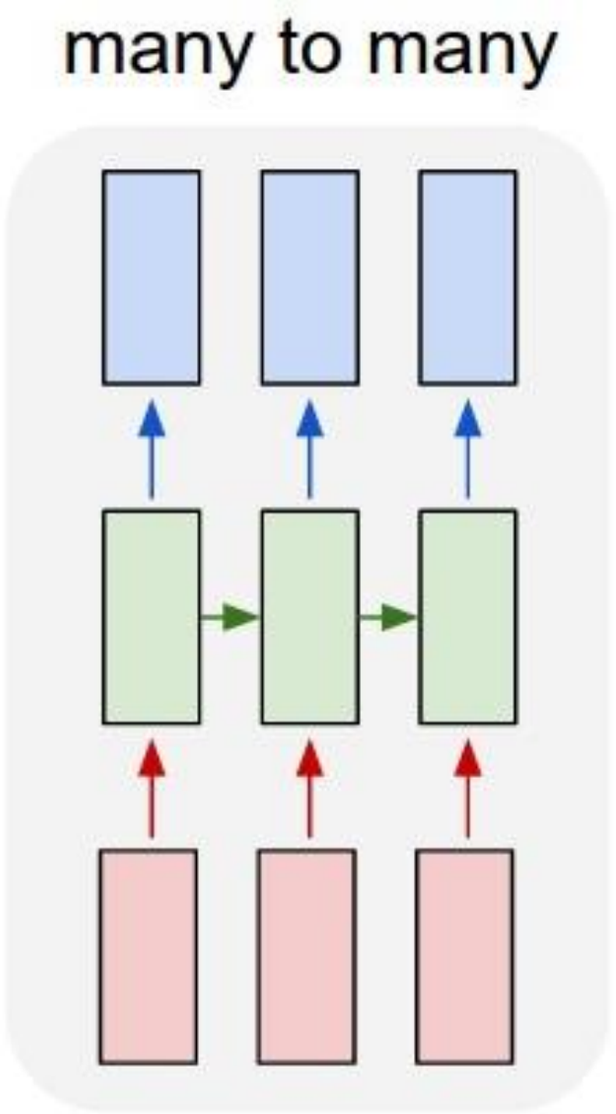
Model

```
z = model(x):
    h = Convolution2D((5,5),filt=8, …)(x)
    h = MaxPooling(…)(h)
    h = Convolution2D ((5,5),filt=16, …)((h)
    h = MaxPooling(…)(h)
    r = Dense(output_classes, act= None)(h)
    return r
```

An Sequence Example (many to many + 1:1)

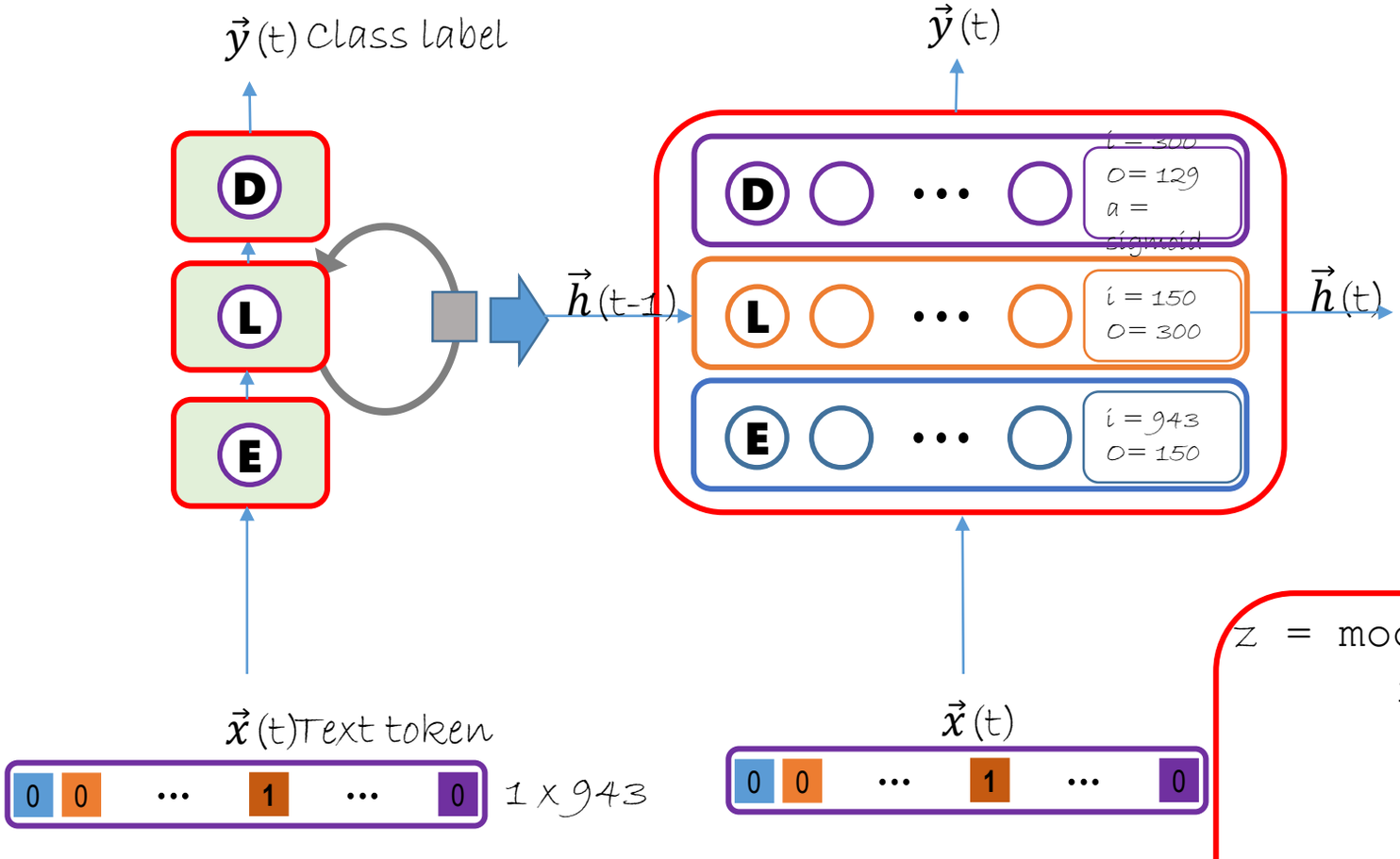
Problem: Tagging entities in Air Traffic Controller (ATIS) data





http://karpathy.github.io/2015/05/21/rnn-effectiveness/

Prepare Network: Recurrent Neural Network



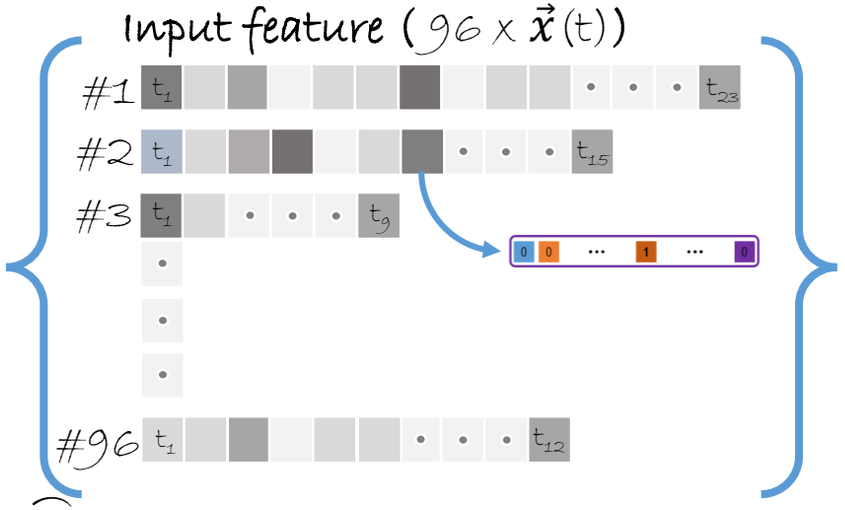

```
z = model():
      return
        Sequential([
            Embedding(emb_dim=150),
            Recurrence(LSTM(hidden_dim=300),
                       go_backwards=False),
            Dense(num_labels = 129)
        ])
```

Prepare Learner

- Many built-in learners
  - SGD, SGD with momentum, Adagrad, RMSProp, Adam, Adamax, AdaDelta, etc.
- Specify learning rate schedule and momentum schedule
- If wanted, specify minibatch size schedule

```
lr_schedule = C.learning_rate_schedule([0.05]*3 + [0.025]*2 + [0.0125],
                        minibatch_size=C.learners.IGNORE, epoch_size=100)
sgd_learner = C.sgd(z.parameters, lr_schedule)
```

Overall Train Workflow




```
z = model():
        return
            Sequential([
                Embedding(emb_dim=150),
                Recurrence(LSTM(hidden_dim=300),
                                go_backwards=False),
                Dense(num_labels = 129)
            ])
```

ATIS Train

96 samples (mini-batch)

One-hot encoded Label (Y: 96 x 129/sample Or word in sequence)

Loss

```
cross_entropy_with_softmax(z,Y)
```

Error

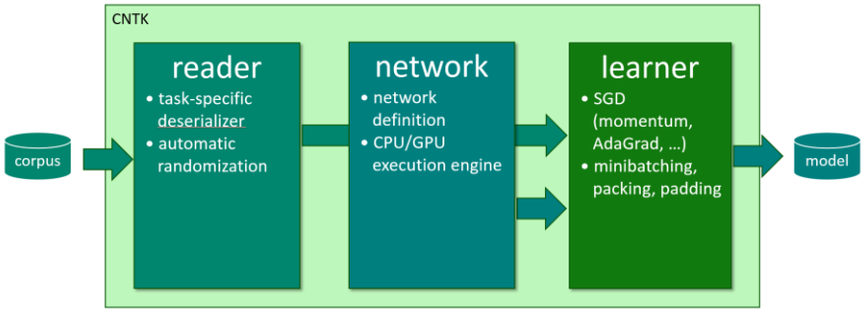
```
classification_error(z,Y)
```

Choose a learner
(SGD, Adam, adagrad etc.)

```
Trainer(model,  (loss, error),  learner)
```

```
Trainer.train_minibatch({X,  Y})
```

Distributed training



- Prepare data
- Configure reader, network, learner (Python)
- Train: -- distributed*!*

```
mpiexec --np 16 --hosts server1,server2,server3,server4    \
python my_cntk_script.py
```

Extensibility: Custom Layer with Built-in Ops

```
def concat_elu(x):
    """ like concatenated ReLU (http://arxiv.org/abs/1603.05201), but then with ELU """
    return cntk.elu(cntk.splice(x, -x, axis=0))

def selu(x, scale, alpha):
    return cntk.element(scale, cntk.element_select(cntk.less(x, 0), alpha * cntk.elu(x), x))


def log_prob_from_logits(x, axis):
    """ numerically stable log_softmax implementation that prevents overflow """
    m = cntk.reduce_max(x, axis)
    return x - m - cntk.log(cntk.reduce_sum(cntk.exp(x-m), axis=axis))
```

Extensibility: Custom Layer with Pure Python

```
class MySigmoid(UserFunction):
    def __init__(self, arg, name='MySigmoid'):
        super(MySigmoid, self).__init__([arg], name=name)

    def forward(self, argument, device=None, outputs_to_retain=None):
        sigmoid_x = 1/(1+numpy.exp(-argument))
        return sigmoid_x, sigmoid_x

    def backward(self, state, root_gradients):
        sigmoid_x = state
        return root_gradients * sigmoid_x * (1 - sigmoid_x)

    def infer_outputs(self):
        return [cntk.output_variable(self.inputs[0].shape,
                 self.inputs[0].dtype, self.inputs[0].dynamic_axes)]
```

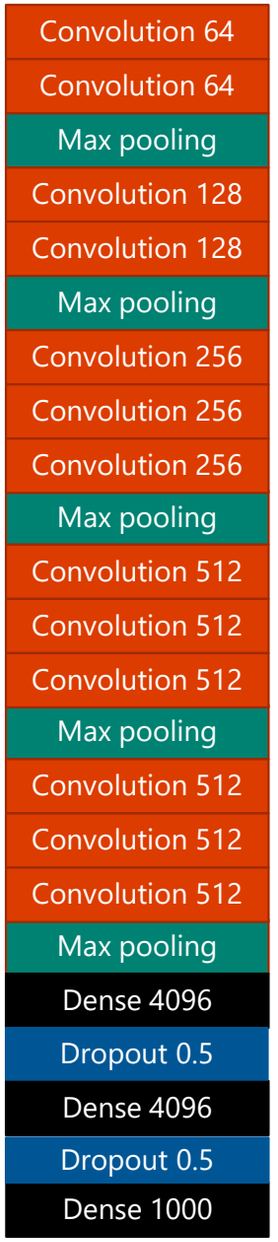
Extensibility: Custom Learner

```
def my_rmsprop(parameters, gradients):
    rho = 0.999
    lr = 0.01
    # We use the following accumulator to store the moving average of every squared gradient
    accumulators = [C.constant(1e-6, shape=p.shape, dtype=p.dtype) for p in parameters]
    update_funcs = []
    for p, g, a in zip(parameters, gradients, accumulators):
        # We declare that `a` will be replaced by an exponential moving average of squared gradients
        # The return value is the expression rho * a + (1-rho) * g * g
        accum_new = cntk.assign(a, rho * a + (1-rho) * g * g)
        # This is the rmsprop update.
        # We need to use accum_new to create a dependency on the assign statement above.
        # This way, when we run this network both assigns happen.
        update_funcs.append(cntk.assign(p, p - lr * g / cntk.sqrt(accum_new)))
    return cntk.combine(update_funcs)

my_learner = cntk.universal(my_rmsprop, z.parameters)
```

https://github.com/Microsoft/CNTK/blob/master/Manual/Manual_How_to_use_learners.ipynb

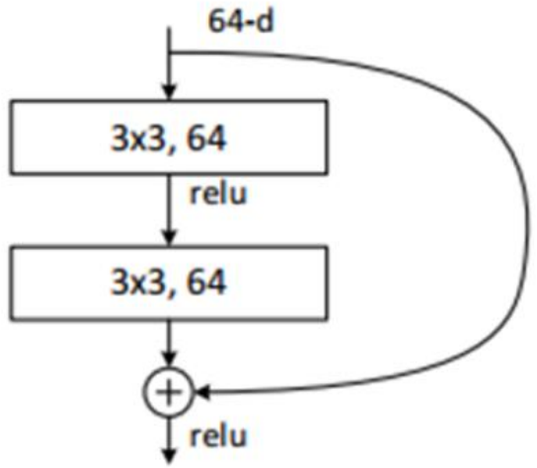
VGG16

```
with C.layers.default_options(activation=C.relu, init=C.glorot_uniform()):
 return C.layers.Sequential([
            C.layers.For(range(2), lambda i: [
                C.layers.Convolution((3,3), [64,128][i], pad=True),
                C.layers.Convolution((3,3), [64,128][i], pad=True),
                C.layers.MaxPooling((3,3), strides=(2,2))
            ]),
             C.layers.For(range(3), lambda i: [
                C.layers.Convolution((3,3), [256,512,512][i], pad=True),
                C.layers.Convolution((3,3), [256,512,512][i], pad=True),
                C.layers.Convolution((3,3), [256,512,512][i], pad=True),
                C.layers.MaxPooling((3,3), strides=(2,2))
            ]),
            C.layers.For(range(2), lambda : [
                C.layers.Dense(4096),
                C.layers.Dropout(0.5)
            ]),
            C.layers.Dense(out_dims, None)])(input)
```



Residual Network (ResNet)

```
def conv_bn(input, filter_size, num_filters, strides=(1,1), activation=C.relu):
    c = Convolution(filter_size, num_filters, None)(input)
    r = BatchNormalization(…)(c)
    if activation != None:
        r = activation(r)
    return r

def resnet_basic(input, num_filters):
    c1 = conv_bn(input, (3,3), num_filters)
    c2 = conv_bn(c1, (3,3), num_filters, activation=None)
    return C.relu(c2 + input)

def resnet_basic_inc(input, num_filters, strides=(2,2)):
    c1 = conv_bn (input, (3,3), num_filters, strides)
    c2 = conv_bn(c1, (3,3), num_filters, activation=None)
    s  = conv_bn(input, (1,1), num_filters, strides, None)
    p  = c2 + s
    return relu(p)
```



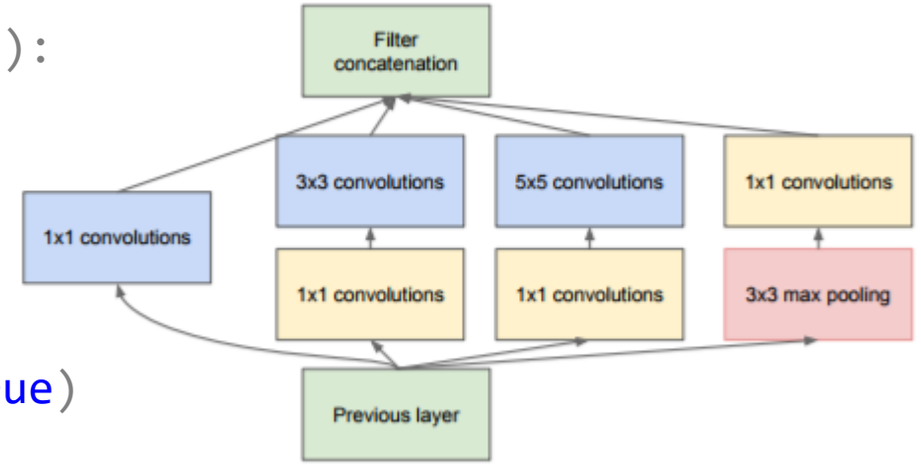
Residual Network (ResNet)

```
def create_resnet_model(input, out_dims):
    c = conv_bn(input, (3,3), 16)
    r1_1 = resnet_basic_stack(c, 16, 3)

    r2_1 = resnet_basic_inc(r1_1, 32)
    r2_2 = resnet_basic_stack(r2_1, 32, 2)

    r3_1 = resnet_basic_inc(r2_2, 64)
    r3_2 = resnet_basic_stack(r3_1, 64, 2)

    pool = C.layers.GlobalAveragePooling()(r3_2)
    net = C.layers.Dense(out_dims,
                         C.he_normal(),
                         None)(pool)
```

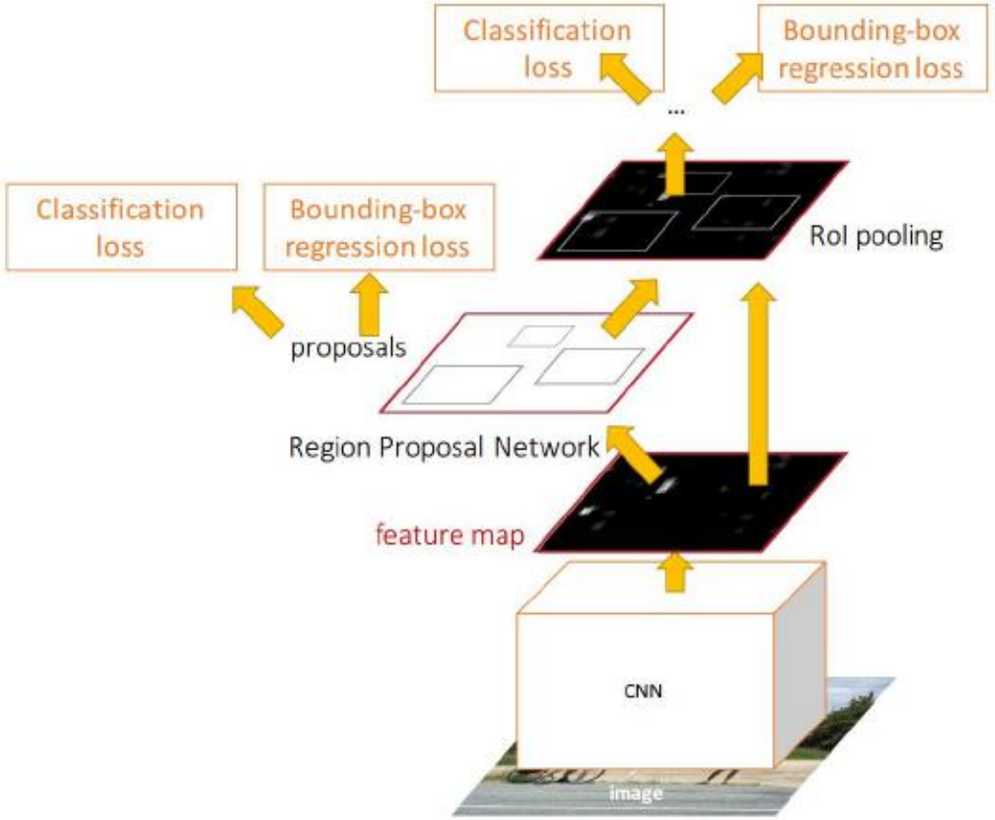
```
def resnet_basic_stack(input,
                       num_filters,
                       num_stack):
    r = input
    for _ in range(num_stack):
        r = resnet_basic(r, num_filters)
    return r
```

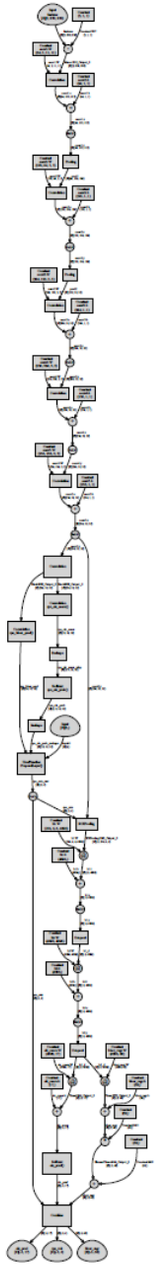
Inception Network

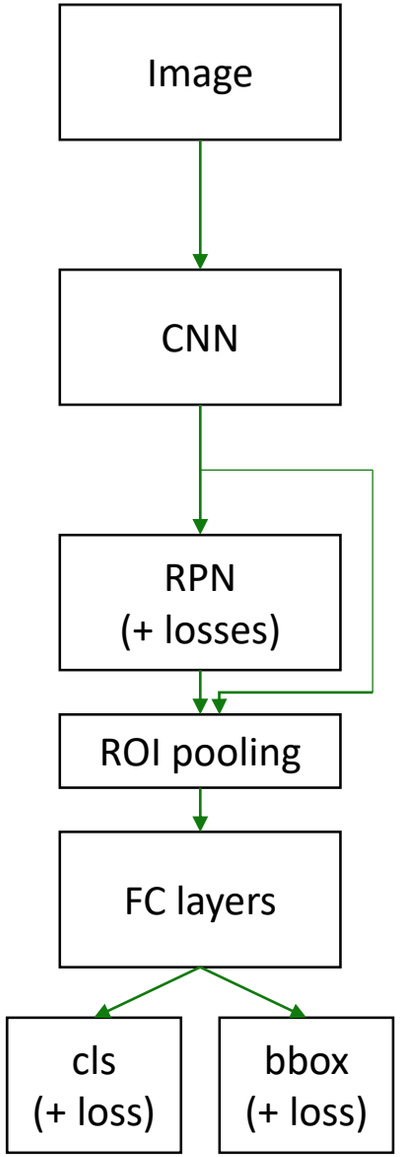
```
inception_block(input, num1x1, num3x3, num5x5, num_pool):
    # 1x1 Convolution
    branch1x1 = conv_bn(input, num1x1, (1,1), True)

    # 3x3 Convolution
    branch3x3 = conv_bn(input, num3x3[0], (1,1), True)
    branch3x3 = conv_bn(branch3x3, num3x3[1], (3,3), True)

    # 5x5 Convolution
    branch5x5 = conv_bn(input, num5x5[0], (1,1), True)
    branch5x5 = conv_bn(branch5x5, num5x5[1], (5,5), True)

    # Max pooling
    branch_pool = C.layers.MaxPooling((3,3), True)(input)
    branch_pool = conv_bn(branch_pool, num_pool, (1,1), True)

    return C.splice(branch1x1, branch3x3, branch5x5, branch_pool)
```



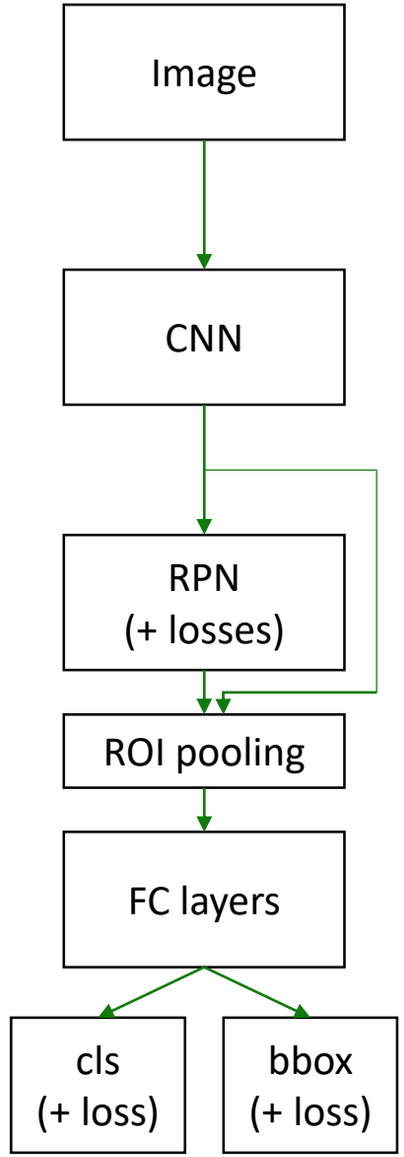
Faster R-CNN in CNTK



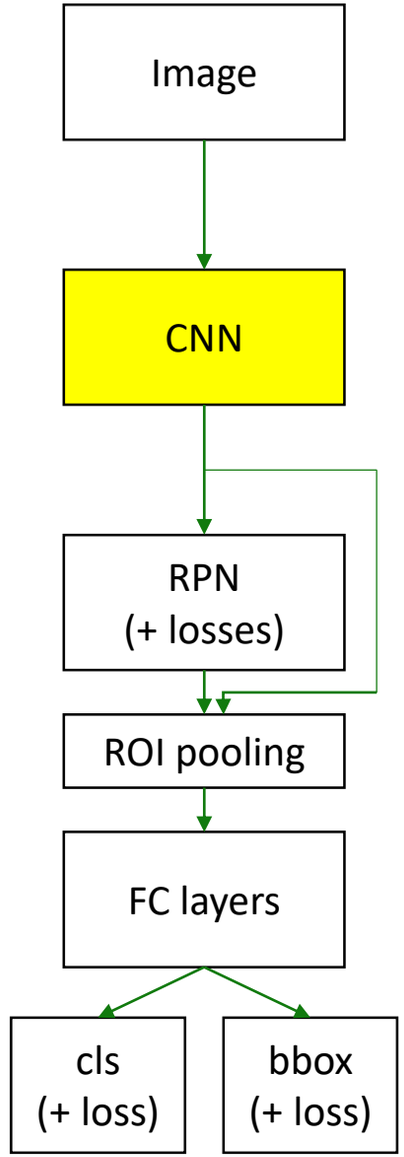




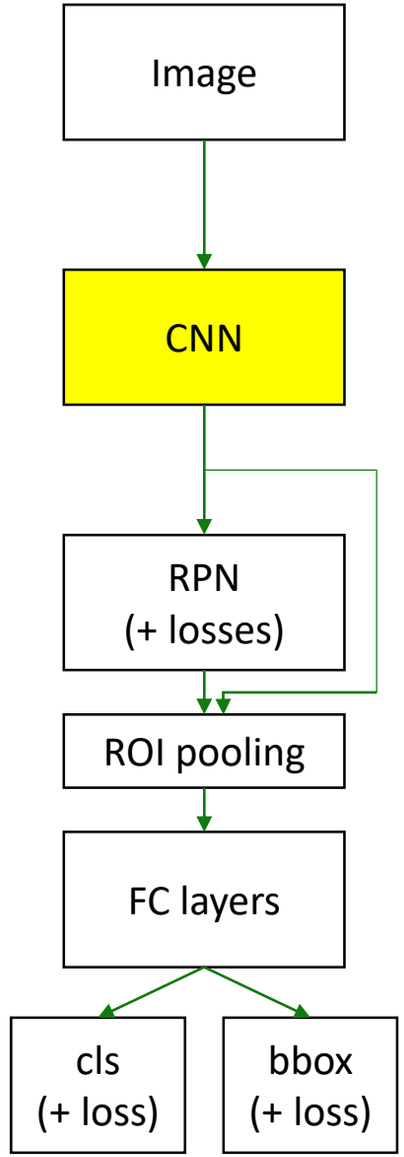
```
def create_faster_rcnn_predictor(input_var, gt_boxes, img_dims):

    base_model  = load_model(base_model_file)
    conv_layers = clone_model(base_model, ['data'], ['relu5_3'], CloneMethod.freeze)

    conv_out = conv_layers(input_var)

    rpn_rois, rpn_losses = create_rpn(conv_out, gt_boxes, img_dims)

    roi_out   = roipooling(conv_out, rpn_rois, cntk.MAX_POOLING, (roi_dim, roi_dim))

    fc_layers   = clone_model(base_model, ['pool5'], ['drop7'], CloneMethod.clone)
    fc_out    = fc_layers(roi_out)

    cls_score = Dense(shape=(4096, num_classes), None)(fc_out)
    bbox_pred = Dense(shape=(4096, num_classes*4), None)(fc_out)

    det_losses = create_detector_losses(cls_score, bbox_pred, rpn_rois, gt_boxes)
    loss = rpn_losses + det_losses
    pred_error = classification_error(cls_score, label_targets, axis=1)

    return loss, pred_error
```



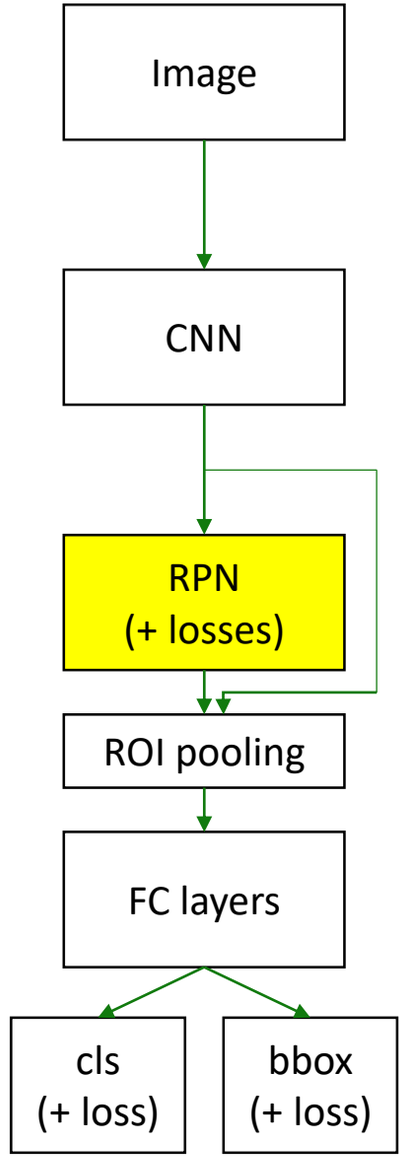
```
def create_faster_rcnn_predictor(input_var, gt_boxes, img_dims):

    base_model  = load_model(base_model_file)
    conv_layers = clone_model(base_model, ['data'], ['relu5_3'], CloneMethod.freeze)

    conv_out = conv_layers(input_var)

    rpn_rois, rpn_losses = create_rpn(conv_out, gt_boxes, img_dims)

    roi_out   = roipooling(conv_out, rpn_rois, cntk.MAX_POOLING, (roi_dim, roi_dim))

    fc_layers   = clone_model(base_model, ['pool5'], ['drop7'], CloneMethod.clone)
    fc_out    = fc_layers(roi_out)

    cls_score = Dense(shape=(4096, num_classes), None)(fc_out)
    bbox_pred = Dense(shape=(4096, num_classes*4), None)(fc_out)

    det_losses = create_detector_losses(cls_score, bbox_pred, rpn_rois, gt_boxes)
    loss = rpn_losses + det_losses
    pred_error = classification_error(cls_score, label_targets, axis=1)

    return loss, pred_error
```



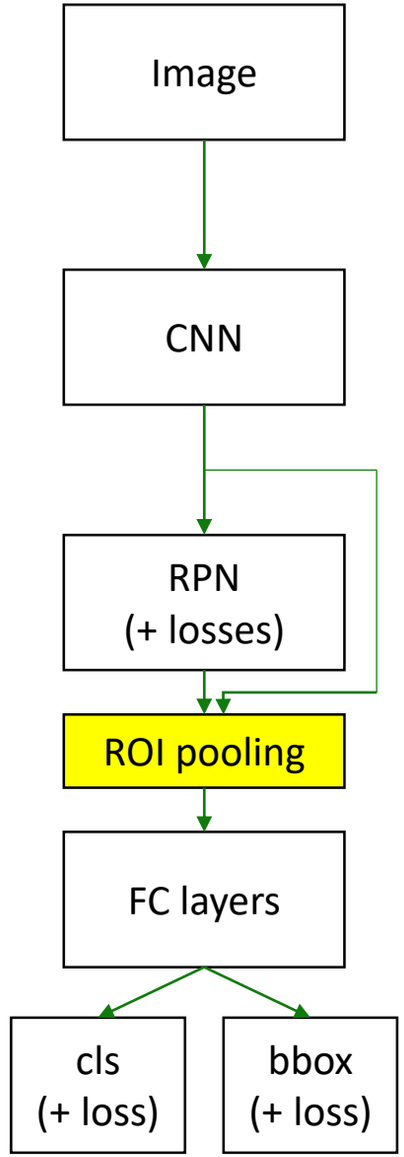
```
def create_faster_rcnn_predictor(input_var, gt_boxes, img_dims):

    base_model   = load_model(base_model_file)
    conv_layers = clone_model(base_model, ['data'], ['relu5_3'], CloneMethod.freeze)

    conv_out = conv_layers(input_var)

    rpn_rois, rpn_losses = create_rpn(conv_out, gt_boxes, img_dims)

    roi_out    = roipooling(conv_out, rpn_rois, cntk.MAX_POOLING, (roi_dim, roi_dim))

    fc_layers    = clone_model(base_model, ['pool5'], ['drop7'], CloneMethod.clone)
    fc_out     = fc_layers(roi_out)

    cls_score = Dense(shape=(4096, num_classes), None)(fc_out)
    bbox_pred = Dense(shape=(4096, num_classes*4), None)(fc_out)

    det_losses = create_detector_losses(cls_score, bbox_pred, rpn_rois, gt_boxes)
    loss = rpn_losses + det_losses
    pred_error = classification_error(cls_score, label_targets, axis=1)

    return loss, pred_error
```



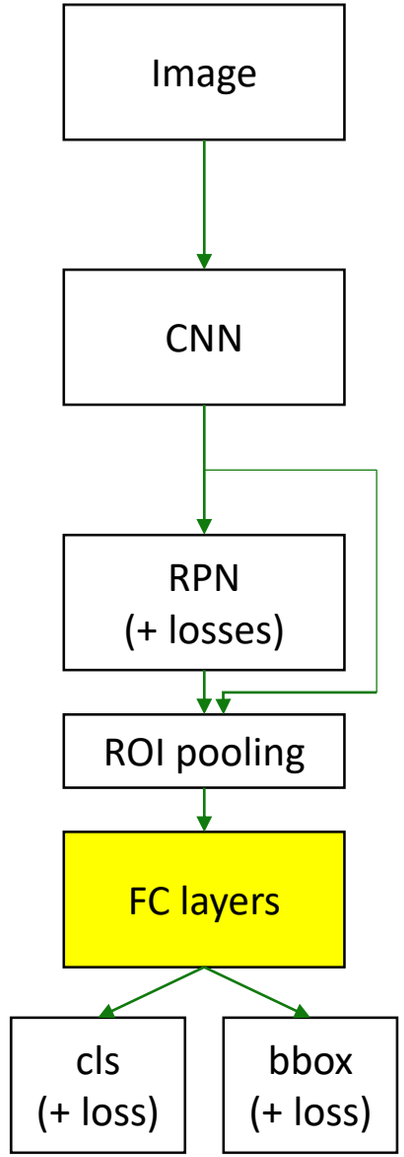
```
def create_faster_rcnn_predictor(input_var, gt_boxes, img_dims):

    base_model   = load_model(base_model_file)
    conv_layers = clone_model(base_model, ['data'], ['relu5_3'], CloneMethod.freeze)

    conv_out = conv_layers(input_var)

    rpn_rois, rpn_losses = create_rpn(conv_out, gt_boxes, img_dims)

    roi_out    = roipooling(conv_out, rpn_rois, cntk.MAX_POOLING, (roi_dim, roi_dim))

    fc_layers    = clone_model(base_model, ['pool5'], ['drop7'], CloneMethod.clone)
    fc_out     = fc_layers(roi_out)

    cls_score = Dense(shape=(4096, num_classes), None)(fc_out)
    bbox_pred = Dense(shape=(4096, num_classes*4), None)(fc_out)

    det_losses = create_detector_losses(cls_score, bbox_pred, rpn_rois, gt_boxes)
    loss = rpn_losses + det_losses
    pred_error = classification_error(cls_score, label_targets, axis=1)

    return loss, pred_error
```



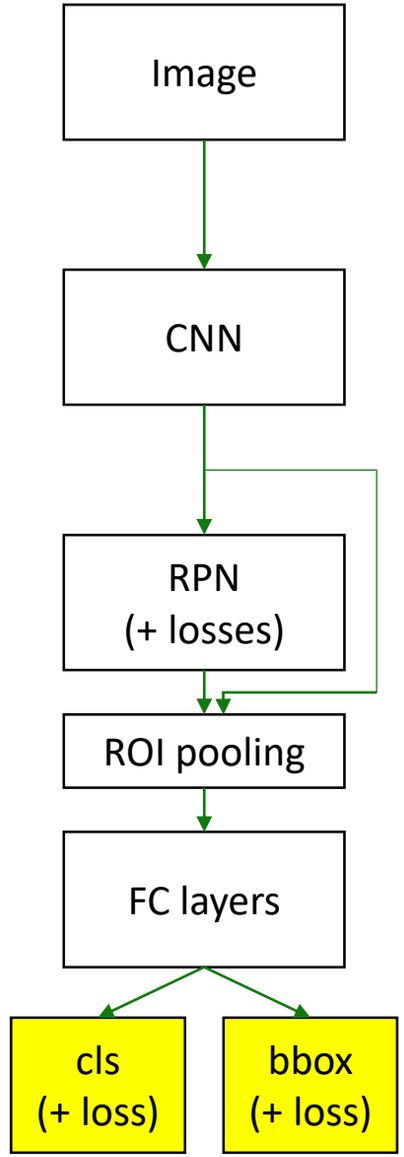
```
def create_faster_rcnn_predictor(input_var, gt_boxes, img_dims):

    base_model   = load_model(base_model_file)
    conv_layers = clone_model(base_model, ['data'], ['relu5_3'], CloneMethod.freeze)

    conv_out = conv_layers(input_var)

    rpn_rois, rpn_losses = create_rpn(conv_out, gt_boxes, img_dims)

    roi_out   = roipooling(conv_out, rpn_rois, cntk.MAX_POOLING, (roi_dim, roi_dim))

    fc_layers   = clone_model(base_model, ['pool5'], ['drop7'], CloneMethod.clone)
    fc_out     = fc_layers(roi_out)

    cls_score = Dense(shape=(4096, num_classes), None)(fc_out)
    bbox_pred = Dense(shape=(4096, num_classes*4), None)(fc_out)

    det_losses = create_detector_losses(cls_score, bbox_pred, rpn_rois, gt_boxes)
    loss = rpn_losses + det_losses
    pred_error = classification_error(cls_score, label_targets, axis=1)

    return loss, pred_error
```



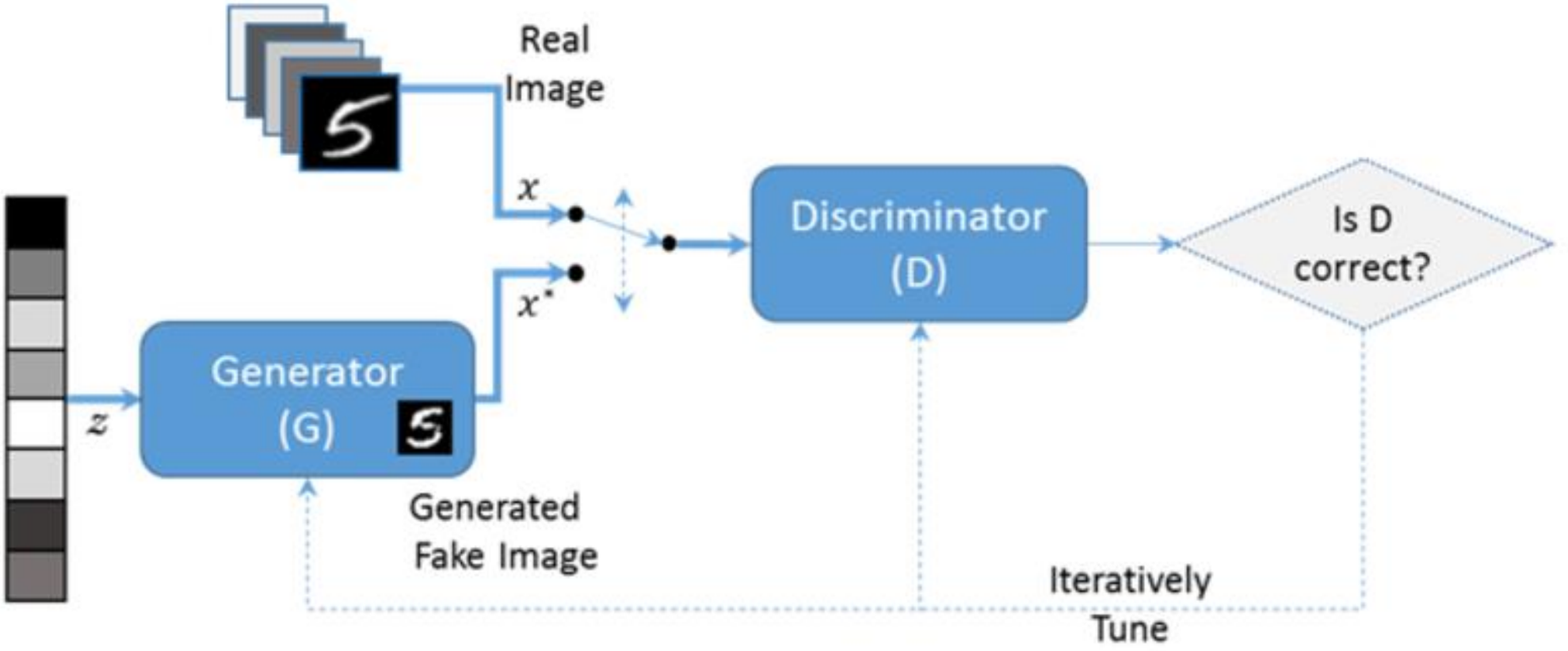
```
def create_faster_rcnn_predictor(input_var, gt_boxes, img_dims):

    base_model   = load_model(base_model_file)
    conv_layers = clone_model(base_model, ['data'], ['relu5_3'], CloneMethod.freeze)

    conv_out = conv_layers(input_var)

    rpn_rois, rpn_losses = create_rpn(conv_out, gt_boxes, img_dims)

    roi_out    = roipooling(conv_out, rpn_rois, cntk.MAX_POOLING, (roi_dim, roi_dim))

    fc_layers    = clone_model(base_model, ['pool5'], ['drop7'], CloneMethod.clone)
    fc_out     = fc_layers(roi_out)

    cls_score = Dense(shape=(4096, num_classes), None)(fc_out)
    bbox_pred = Dense(shape=(4096, num_classes*4), None)(fc_out)

    det_losses = create_detector_losses(cls_score, bbox_pred, rpn_rois, gt_boxes)
    loss = rpn_losses + det_losses
    pred_error = classification_error(cls_score, label_targets, axis=1)

    return loss, pred_error
```



```
def create_faster_rcnn_predictor(input_var, gt_boxes, img_dims):

    base_model   = load_model(base_model_file)
    conv_layers = clone_model(base_model, ['data'], ['relu5_3'], CloneMethod.freeze)

    conv_out = conv_layers(input_var)

    rpn_rois, rpn_losses = create_rpn(conv_out, gt_boxes, img_dims)

    roi_out    = roipooling(conv_out, rpn_rois, cntk.MAX_POOLING, (roi_dim, roi_dim))

    fc_layers    = clone_model(base_model, ['pool5'], ['drop7'], CloneMethod.clone)
    fc_out     = fc_layers(roi_out)

    cls_score = Dense(shape=(4096, num_classes), None)(fc_out)
    bbox_pred = Dense(shape=(4096, num_classes*4), None)(fc_out)

    det_losses = create_detector_losses(cls_score, bbox_pred, rpn_rois, gt_boxes)
    loss = rpn_losses + det_losses
    pred_error = classification_error(cls_score, label_targets, axis=1)

    return loss, pred_error
```



GAN



GAN in CNTK

```
g = generator(z)
d_real = discriminator(real_input)
d_fake = d_real.clone(method='share', substitutions={real_input.output:g.output})

g_loss = 1.0 - C.log(d_fake)
d_loss = -(C.log(d_real) + C.log(1.0 - d_fake))

g_learner = C.adam(parameters=g.parameters,
                   lr=C.learning_rate_schedule(lr, minibatch_size=C.learners.IGNORE),
                   momentum=C.momentum_schedule(momentum))

d_learner = C.adam(parameters=d_real.parameters,
                   lr=C.learning_rate_schedule(lr, minibatch_size=C.learners.IGNORE),
                   momentum=C.momentum_schedule(momentum))

g_trainer = C.Trainer(g, (g_loss, None), g_learner)
d_trainer = C.Trainer(d_real, (d_loss, None), d_learner)
```

MNIST GAN with CNTK

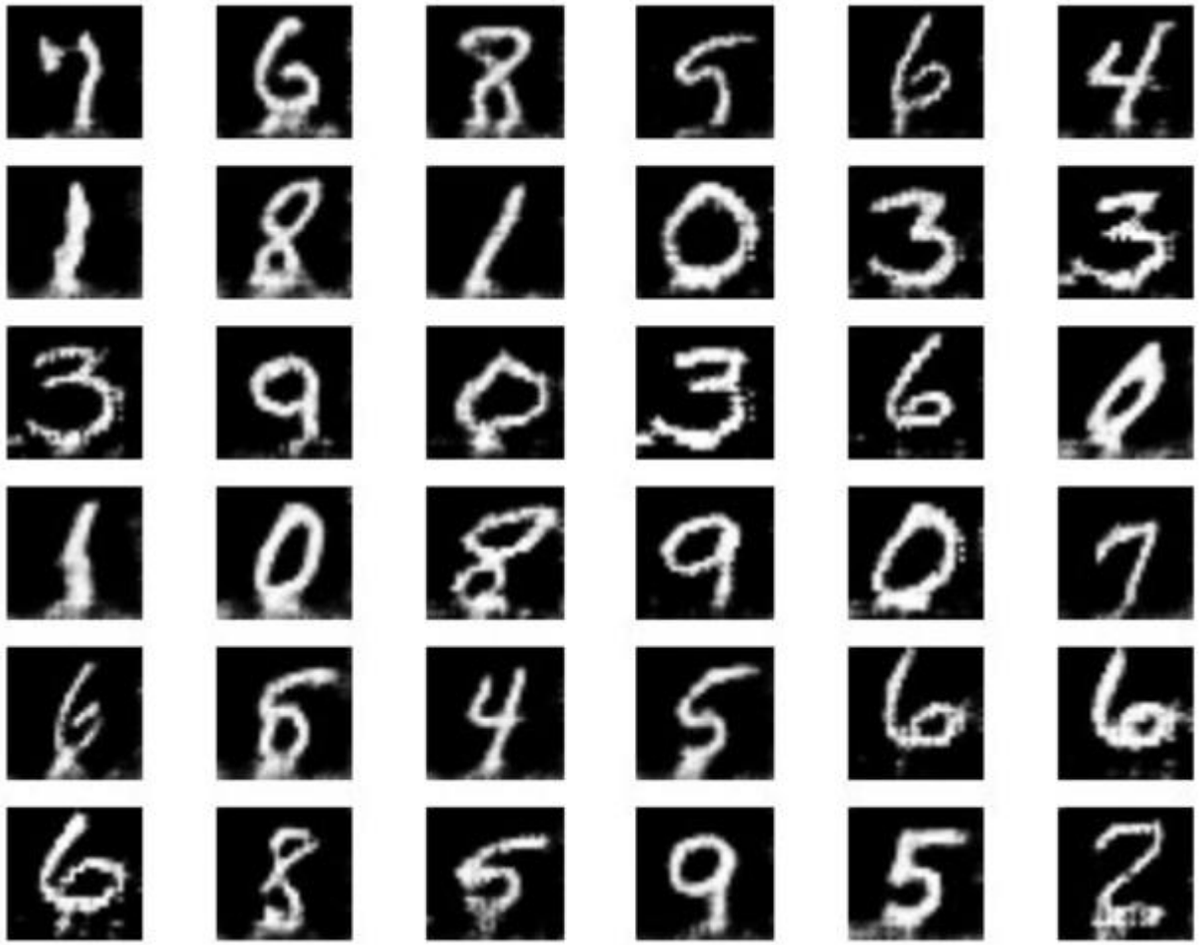
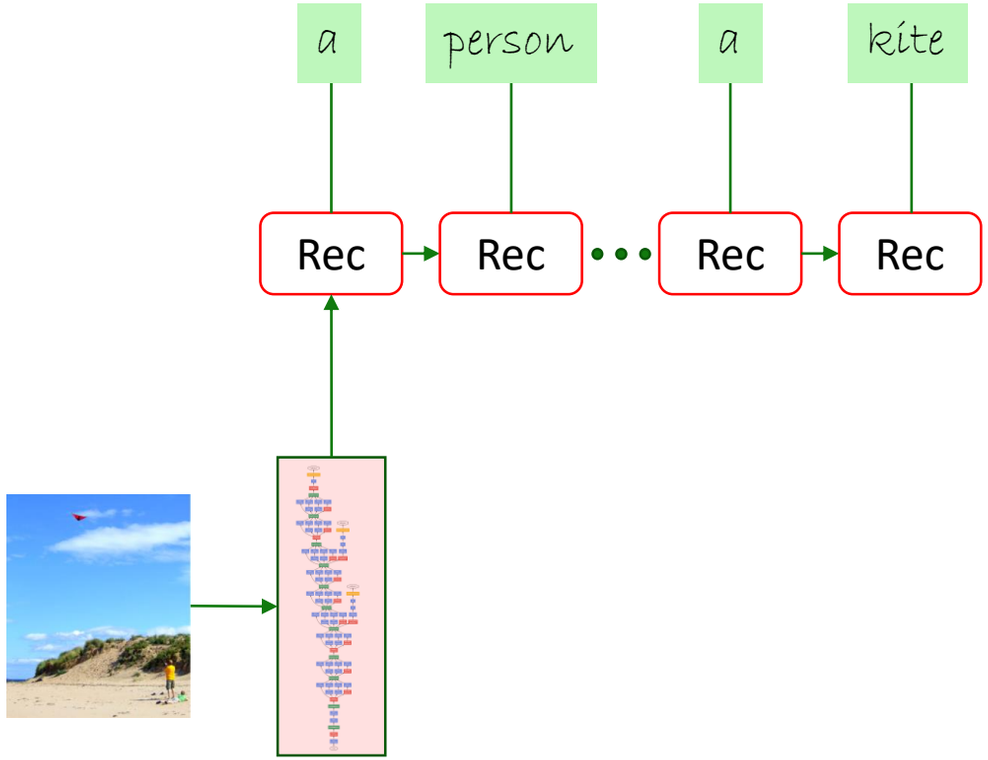
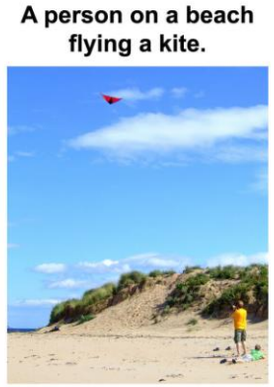
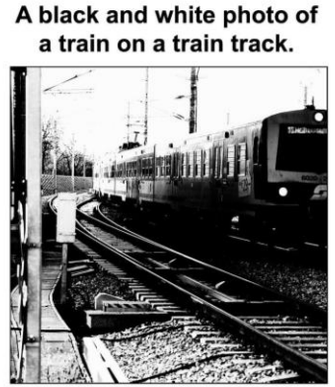
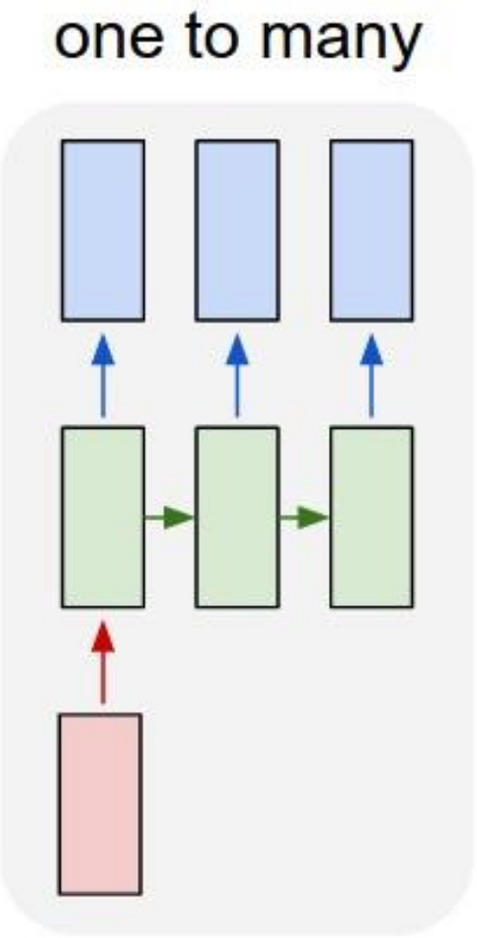


Image Captioning (one to many)







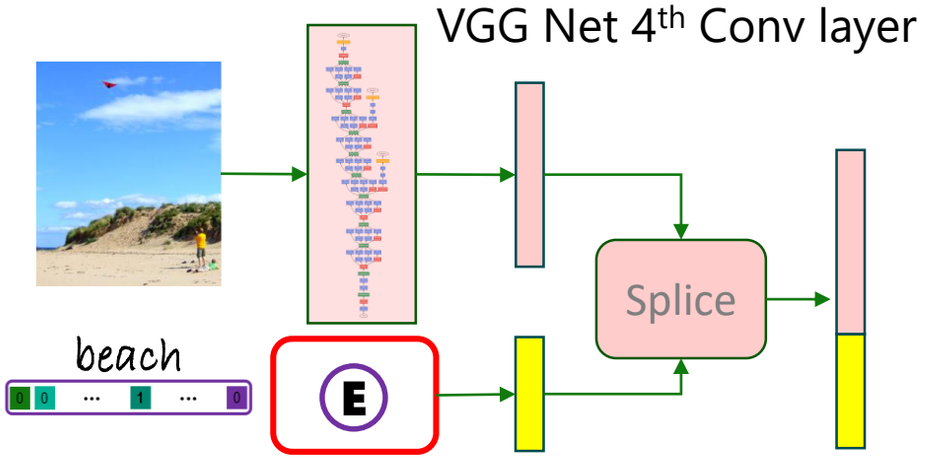


http://karpathy.github.io/2015/05/21/rnn-effectiveness/

Microsoft

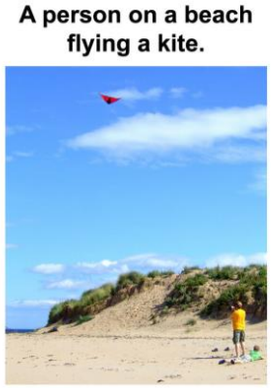
Image Captioning with CNTK

- Input:
  - Image features
  - Word token in the caption

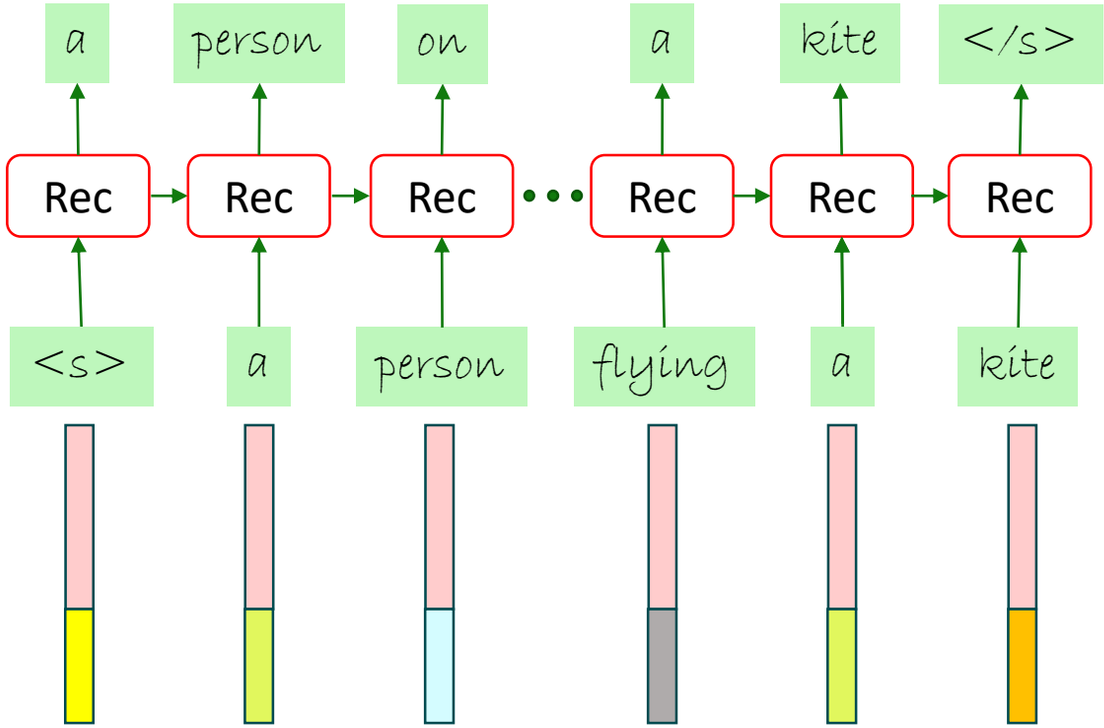



```
# Input image feature
img_fea = VGG_model(img)

# Caption input
cap_in = C.input_variable(shape=(V), is_sparse=True)


img_txt_feature = C.splice(C.reshape(img_fea, …),
                           C.Embedding(EMB_DIM)(cap_in))
```

Image Captioning with CNTK

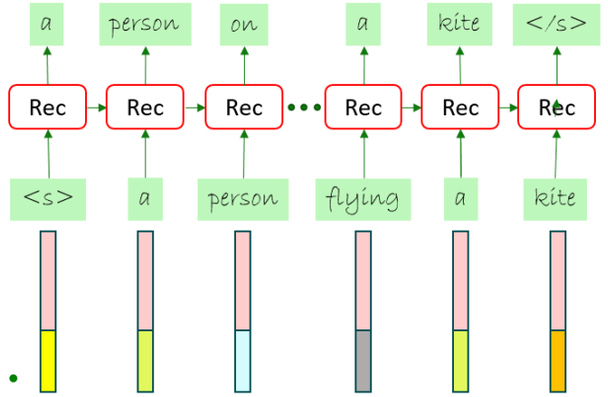


- Use Sequence to Sequence generation machinery
  - Input: Image Feature + word
  - Output: next word




```
img_fea_broadcasted = C.splice(
        C.sequence.broadcast_as(img_fea),
        C.Embedding(EMB_DIM)(cap_in))
```

Image Captioning (Decoder)




```
def eval_greedy(input): # (input*) --> (word sequence*)

    # Decoding is an unfold() operation starting from sentence_start.
    # We must transform (history*, input* -> word_logp*) into
    # a generator (history* -> output*)
    # which holds 'input' in its closure.
    unfold = C.layers.UnfoldFrom(lambda history: model(history, input) >> C.hardmax,
                                 # stop once sentence_end_index is reached
                                 until_predicate=lambda w: w[...,sentence_end_index])

    return unfold(initial_state=sentence_start, dynamic_axes_like=input)
```

https://github.com/Microsoft/CNTK/blob/master/Tutorials/CNTK_204_Sequence_To_Sequence.ipynb

Action Classification

Two main problems:

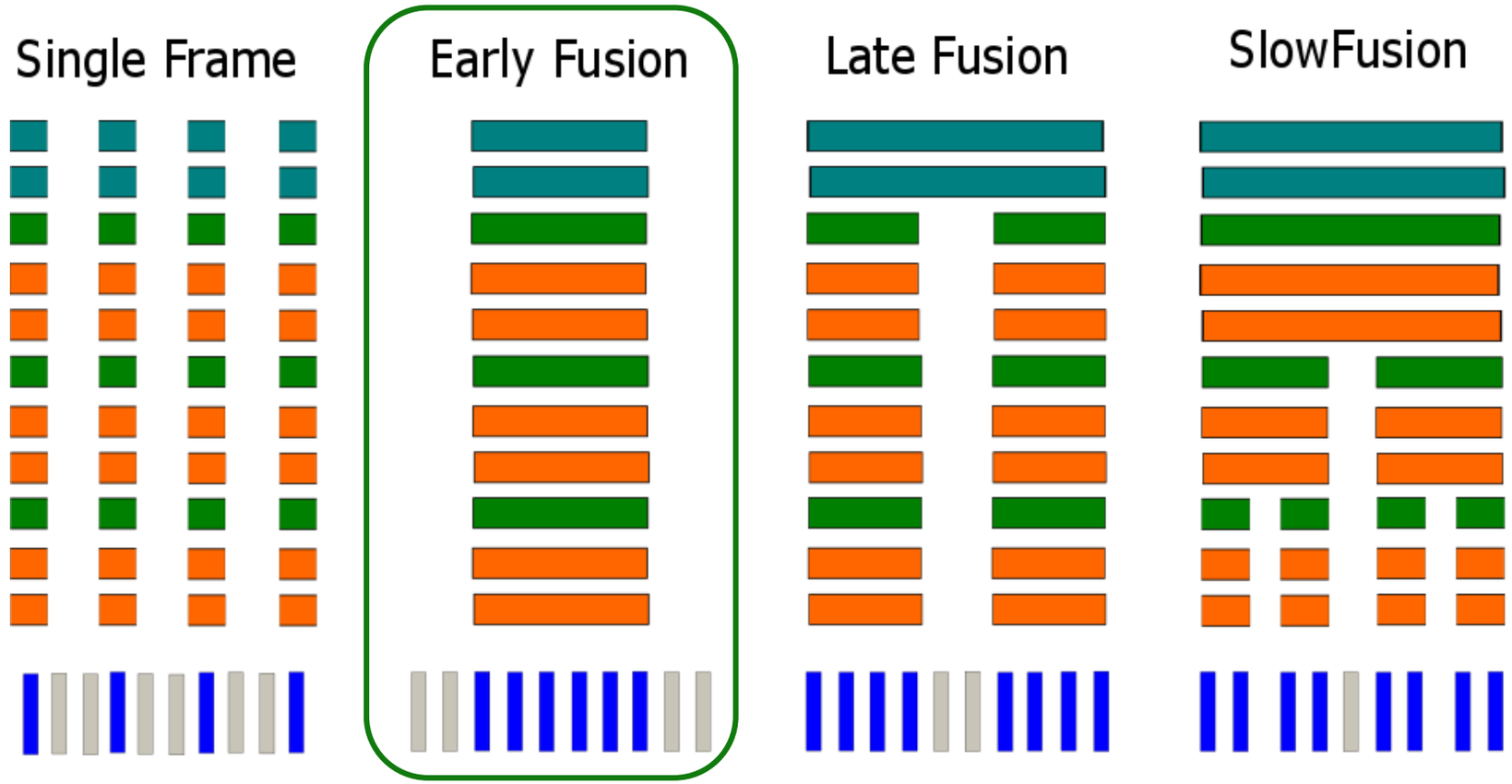
- Action classification
  - Input: a trimmed video clip with a single action.
  - Output: classify the action in the clip.
- Action detection
  - Input: an untrimmed video clip with multiple actions and possible no action.
  - Output: location of each action and its corresponding classification.

Action Classification

Two possible approaches:

- 3D Convolution network
  - Extend 2D convolution into temporal axis.
  - Pick at random a sequence of frames from the video clip.
  - Use the 3D cube as input to the 3D convolution network.
- Pretraining + RNN
  - Use a pretrained model.
  - Extract a feature vector from each frame.
  - Pass the sequence of features into a recurrent network.

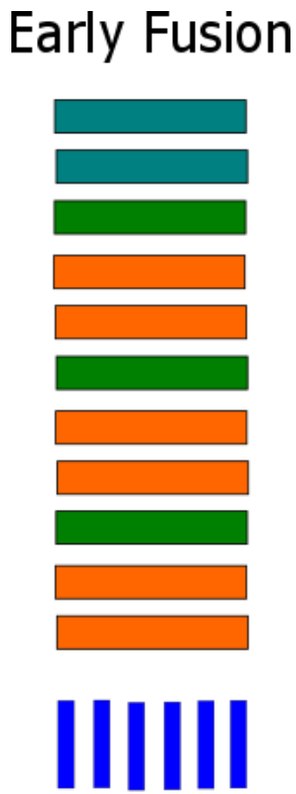
Video Classification Using Feedforward Networks



[Andrej Karpathy, George Toderici, Sanketh Shetty, Thomas Leung, Rahul Sukthankar, Li Fei-Fei, "Large-scale Video Classification with Convolutional Neural Networks", CVPR 2014]

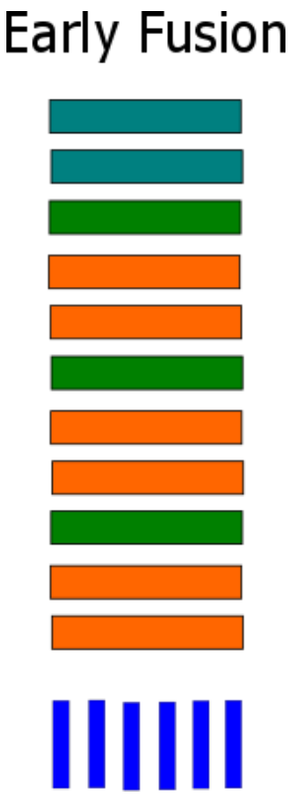
3D Convolution Network

```
input_var = C.input_variable((num_channels, sequence_length, image_height, image_width))

with C.default_options (activation=C.relu):
    z = C.layers.Sequential([
            C.layers.Convolution3D((3,3,3), 64),
            C.layers.MaxPooling((1,2,2), (1,2,2)),
            C.layers.For(range(3), lambda i: [
                C.layers.Convolution3D((3,3,3), [96, 128, 128][i]),
                C.layers.Convolution3D((3,3,3), [96, 128, 128][i]),
                C.layers.MaxPooling((2,2,2), (2,2,2))
            ]),
            C.layers.For(range(2), lambda : [
                C.layers.Dense(1024),
                C.layers.Dropout(0.5)
            ]),
            C.layers.Dense(num_output_classes, activation=None)
        ])(input_var)
```



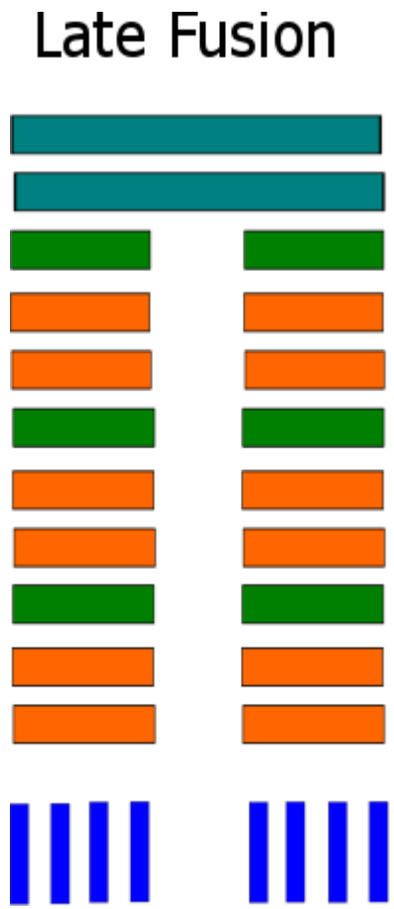
(Pseudo) 3D Convolution Network

```
input_var = C.input_variable((num_channels, sequence_length, image_height, image_width))

with C.default_options (activation=C.relu):
    z = C.layers.Sequential([
            C.layers.Convolution3D((1,3,3), 64),
            C.layers.Convolution3D((3,1,1), 64),
            C.layers.MaxPooling((1,2,2), (1,2,2)),
            C.layers.For(range(3), lambda i: [
                C.layers.Convolution3D((3,3,3), [96, 128, 128][i]),
                C.layers.Convolution3D((3,3,3), [96, 128, 128][i]),
                C.layers.MaxPooling((2,2,2), (2,2,2))
            ]),
            C.layers.For(range(2), lambda : [
                C.layers.Dense(1024),
                C.layers.Dropout(0.5)
            ]),
            C.layers.Dense(num_output_classes, activation=None)
        ])(input_var)
```



Late Fusion

```
with C.default_options (activation=C.relu):
        z1 = C.layers.Sequential([
                C.layers.Convolution3D((3,3,3), 64),
                C.layers.MaxPooling((1,2,2), (1,2,2)),
                C.layers.For(range(3), lambda i: [
                    C.layers.Convolution3D((3,3,3), [96, 128, 128][i]),
                    C.layers.Convolution3D((3,3,3), [96, 128, 128][i]),
                    C.layers.MaxPooling((2,2,2), (2,2,2))
                ]),
            ])(input_var1)
        z2 = C.layers.Sequential([…])(input_var2)
        z  = C.layers.Sequential([
                C.layers.For(range(2), lambda : [
                    C.layers.Dense(1024),
                    C.layers.Dropout(0.5)
                ]),
                C.layers.Dense(num_output_classes, None)
            ])(C.splice(z1, z2, axis=0))
```



Pretraining + RNN

- Loading a pretrained model.
- Extract feature from each frame in a video.
- Feed those frames to LSTM.
- Classify the last output of the LSTM.

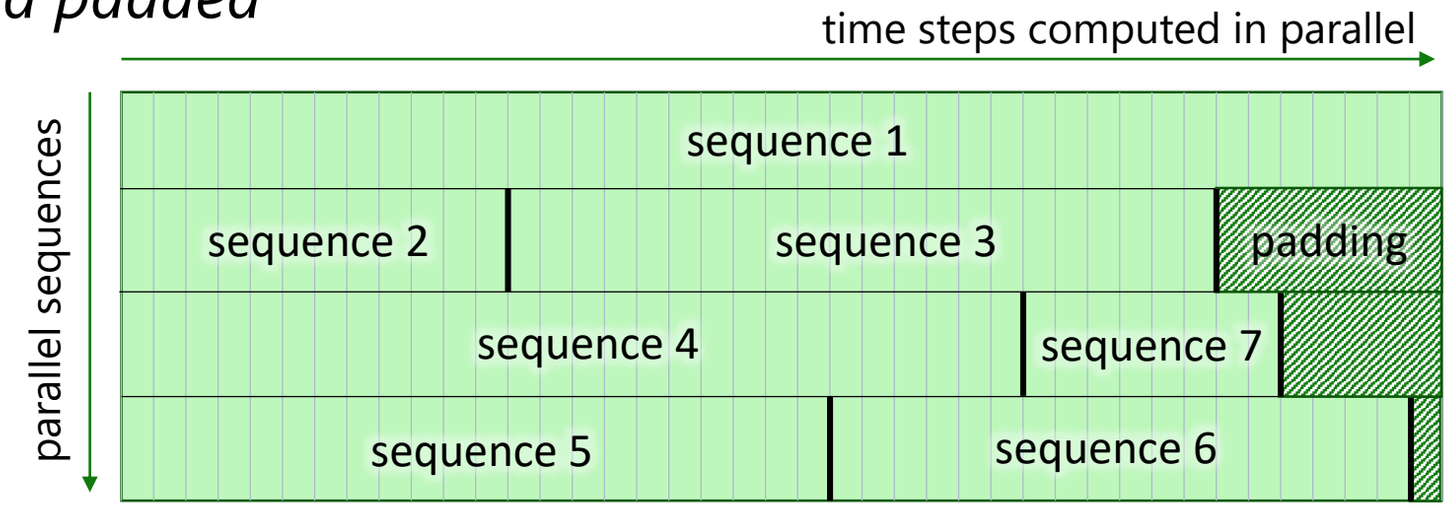
Loading Model and Extract Feature

- Download a pretrained model from CNTK site.
- Convert a pretrained model from another toolkit such as Caffe.
- Train you own network from scratch.
- Loading a model and extract feature is trivial, as shown below:

```
loaded_model  = C.load_model(model_file)
node_in_graph = loaded_model.find_by_name(node_name)
output_node   = C.as_composite(node_in_graph)

output = output_node.eval(<image>)
```

Variable-Length Sequences in CNTK

- Minibatches containing sequences of different lengths are automatically packed *and padded*



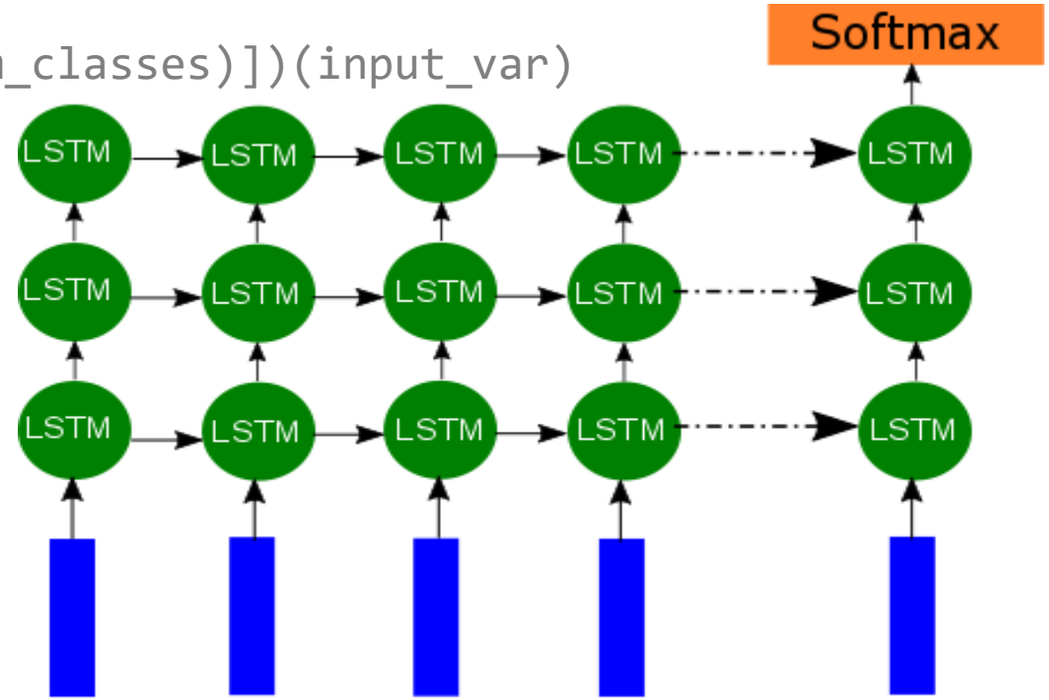
- Fully transparent batching
- Recurrent → CNTK unrolls, handles sequence boundaries
- Non-recurrent operations → parallel
- Sequence reductions → mask

Feature sequence classification

- Use cntk.sequence.input instead of cntk.input.
- Define the network for a single sample.
- No explicit handling of batch or sequence axes.
- Use cntk.sequence.last to get the last item in the sequence.
- Use cntk.sequence.first to get the first item in the sequence, in case of bidirection.

Feature Sequence Classification

```
input_var = C.sequence.input_variable(shape=input_dim)
label_var = C.input_variable(num_classes)

z = C.layers.Sequential([C.For(range(3), lambda : Recurrence(LSTM(hidden_dim))),
                         C.sequence.last,
                         C.layers.Dense(num_classes)])(input_var)
```



Feature Sequence Classification (Bi-Directional)

```
input_var = C.sequence.input_variable(shape=input_dim)
label_var = C.input_variable(num_classes)

fwd = C.layers.Sequential(
          [C.For(range(3), lambda : Recurrence(GRU(hidden_dim))),
           C.sequence.last])(input_var)

bwd = C.layers.Sequential(
          [C.For(range(3), lambda : Recurrence(GRU(hidden_dim), go_backwards=True)),
           C.sequence.first])(input_var)

z = C.layers.Dense(num_classes)(C.splice(fwd, bwd))
```

Conclusions

- Performance
  - Speed: faster than others, 5-10x faster on recurrent networks
  - Accuracy: validated examples/recipes
  - Scalability: few lines of change to scale to thousands of GPUs
  - Built-in readers: efficient distributed readers
- Programmability
  - Powerful C++ library for enterprise users
  - Intuitive and performant Python APIs
  - C#/.NET support for both training and inference
  - Java inference support
  - Extensible via user custom layers, learners, readers, etc.