Tricks for Training Neural Models

(Some slides by Yoav Goldberg, Graham Neubig)

Optimization Choices

- **Adaptive learning rate.**
  - adaptive optimizers such as Adam ([Kingma14]) because they can better handle the complex training dynamics of RNNs
- **Gradient clipping.**
  - Print or plot the gradient norm to see its usual range
  - then scale down gradients that exceeds this range.
  - This prevents spikes in the gradients to mess up the parameters during training.
- **Normalizing the loss.** (To get losses of similar magnitude across datasets)
  - sum the loss terms along the sequence and divide them by the maximum seq length.
  - This makes it easier to reuse hyper parameters between experiments.
  - The loss should be averaged across the batch.
- **Early Stopping**

https://danijar.com/tips-for-training-recurrent-neural-networks/

Network Structure (RNN)

- **Use Gated Recurrent Unit.**
- **Layer normalization.** Adding layer normalization (Ba et al 16) to all linear mappings of the recurrent network speeds up learning
- **Stacked recurrent networks.**
  - Recurrent networks need a quadratic number of weights in their layer size.
  - More efficient to stack two or three smaller layers instead of one big one.
  - Sum the outputs of all layers instead of using only the last one, similar to a ResNet or DenseNet.

https://danijar.com/tips-for-training-recurrent-neural-networks/

Model Parameters (RNN)

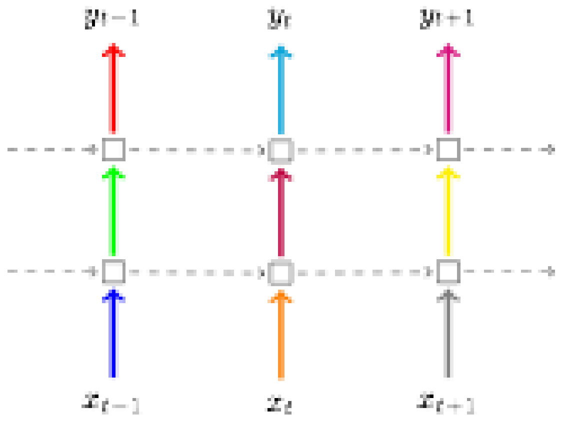
- **Learned initial state.**
  - Initializing the hidden state as zeros → large loss initially
  - Training the initial state as a variable can improve performance as described in https://r2rt.com/non-zero-initial-states-for-recurrent-neural-networks.html
- **Forget gate bias.**
  - It can take a while for a RNN to learn to remember information
  - Initialize biases for LSTM's forget gate to 1 to remember more by default.
  - Similarly, initialize biases for GRU's reset gate to -1.
- **Regularization.** If your model is overfitting, use dropout

https://danijar.com/tips-for-training-recurrent-neural-networks/

Dropout in RNNs

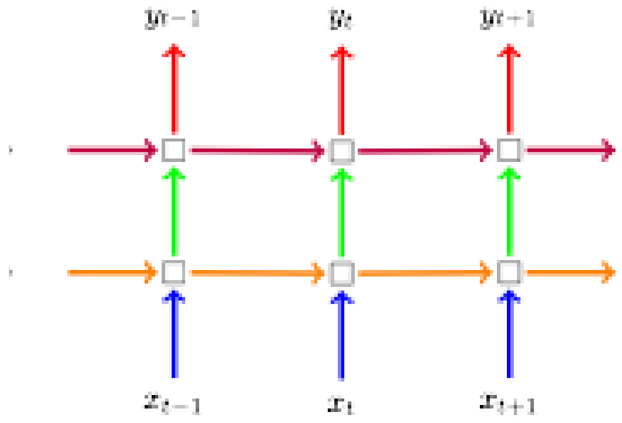
- Still an open question how to perform well.

- One suggestion:



Dropout in RNNs

- Still an open question how to perform well.
- Yarin Gal's Dropout:



Ensembles

- **Same model, different initialization**.
  - Use cross-validation to determine the best hyperparameters,
  - then train multiple models with the best set of hyperparameters but with different random initialization.
  - Suffers from limited variety

- **Top models discovered during cross-validation**.
  - Use cross-validation to determine the best hyperparameters
  - then pick the top few (e.g., 10) models to form the ensemble.
  - Improves the variety of ensemble but has the danger of including suboptimal models

- **Different checkpoints of a single model**.
  - If training is very expensive
  - limited success in taking different checkpoints of a single network over time (for example after every epoch) and using those to form an ensemble.
  - Clearly, this suffers from some lack of variety, but can still work reasonably well in practice.
  - The advantage of this approach is that is very cheap.

http://lamda.nju.edu.cn/weixs/project/CNNTricks/CNNTricks.html

Why are Neural Networks Slow and What Can we Do?

- Big operations, especially for softmaxes over large vocabularies
  - → **Approximate operations or use GPUs**
- GPUs love big operations, but hate doing lots of them
  - → **Reduce the number of operations** through optimized implementations or batching
- Our networks are big, our data sets are big
  - → **Use parallelism** to process many data at once

Approximating Softmax

- Importance Sampling
- Noise Contrastive Estimation
- Simple Negative Sampling
- Hierarchical Softmax

GPUs vs. CPUs

CPU, like a motorcycle

Quick to start, top speed not shabby

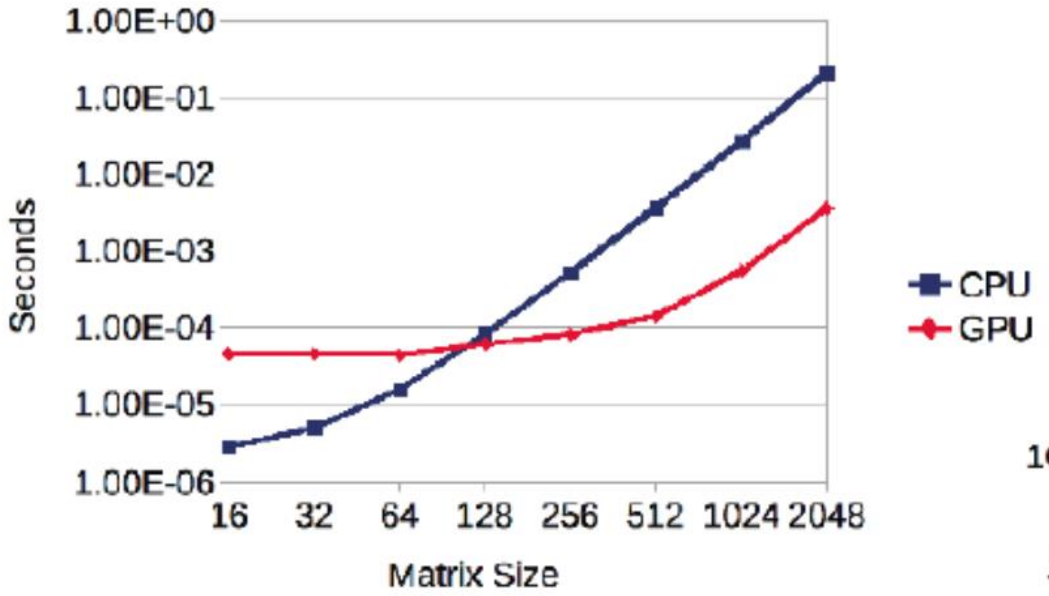
GPU, like an airplane

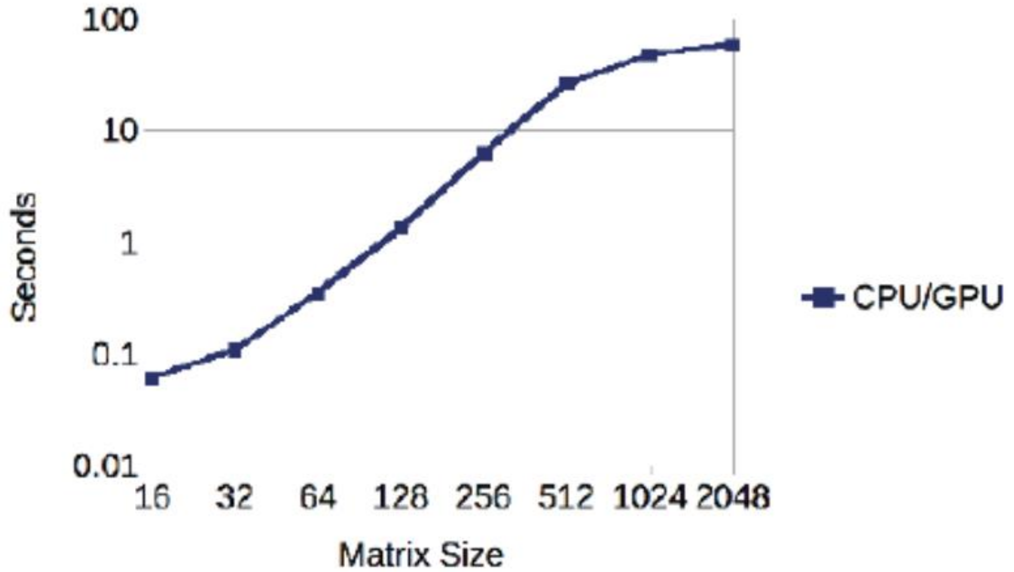
Takes forever to get off the ground, but super-fast once flying

Image Credit: Wikipedia

A Simple Example

- How long does a matrix-matrix multiply take?





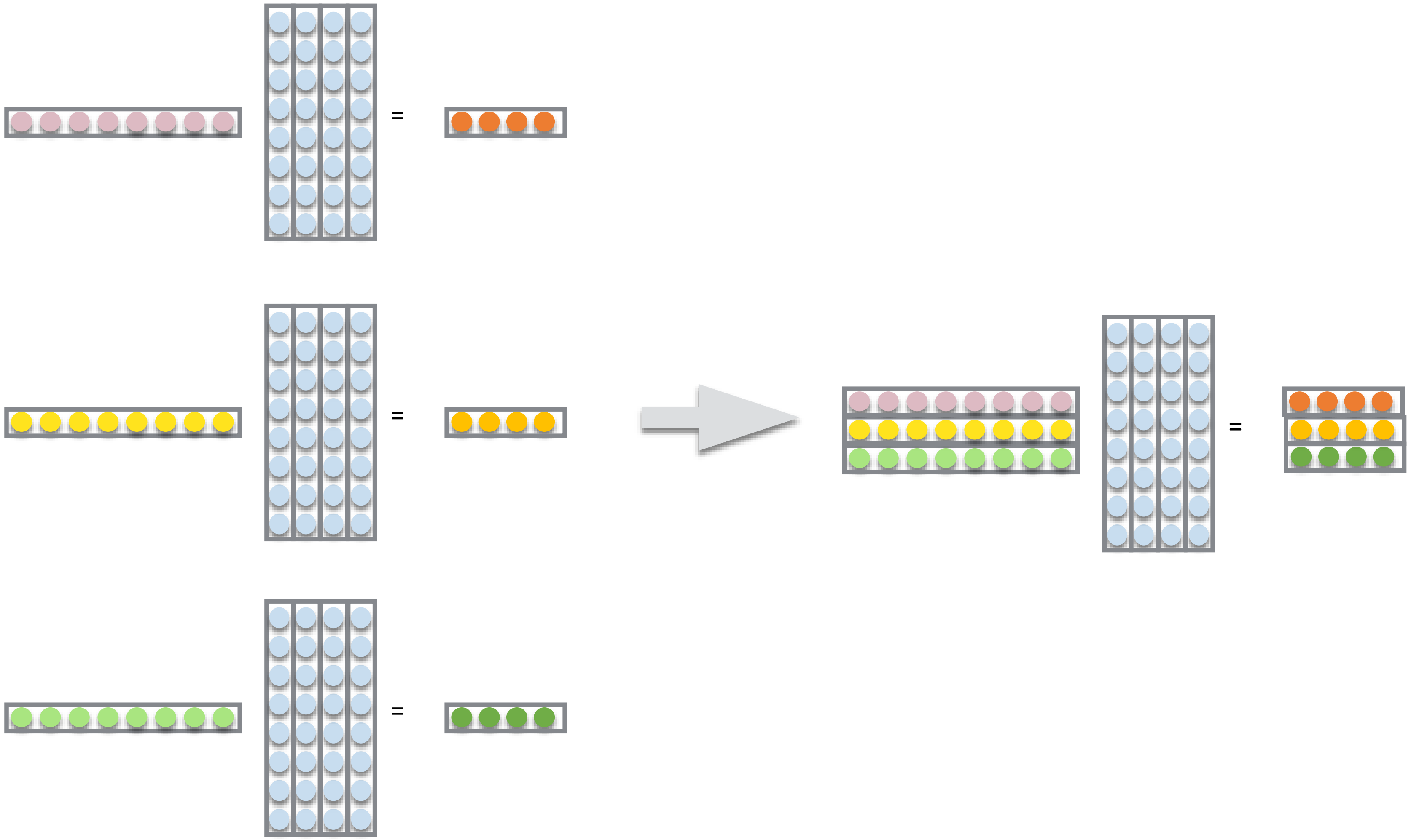
Practically

- Use **CPU for profiling**, it's plenty fast (esp. DyNet) and you can run many more experiments
- For **many applications, CPU is just as fast** or faster than GPU: NLP analysis tasks with small or complicated data/networks
- You see **big gains on GPU when** you have:
  - Very big networks (or softmaxes with no approximation)
  - Do mini-batching
  - Optimize things properly

Batching (in RNNs)

- Most toolkits require a **fixed** computation graph for all examples.
- But RNNs have **different** input lengths. What do we do?
  - Option 1:
    Use a tool that does not pose this limitation.
  - Option 2:
    Enforce max length + 0 padding for shorter sequences.

Batching Reminder

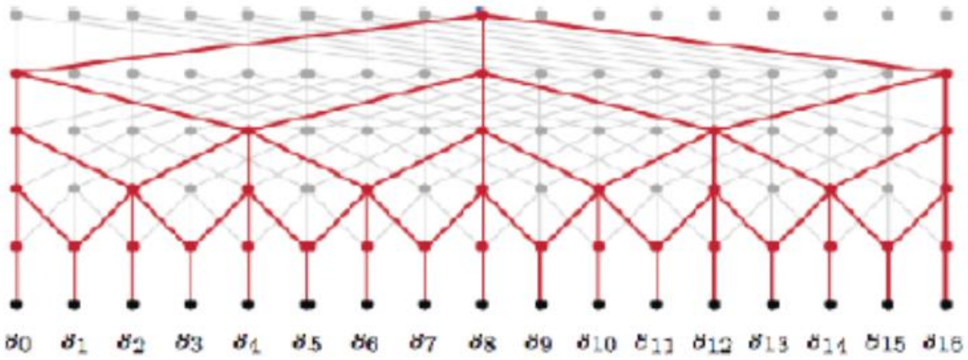


Batching in RNNs

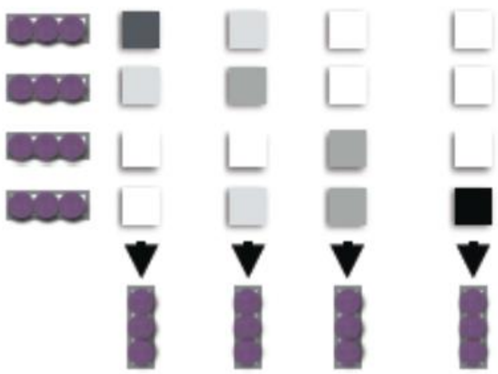
- Sequential in nature, very little parallelism.
- (Compare, e.g., to a Convolutional Network)

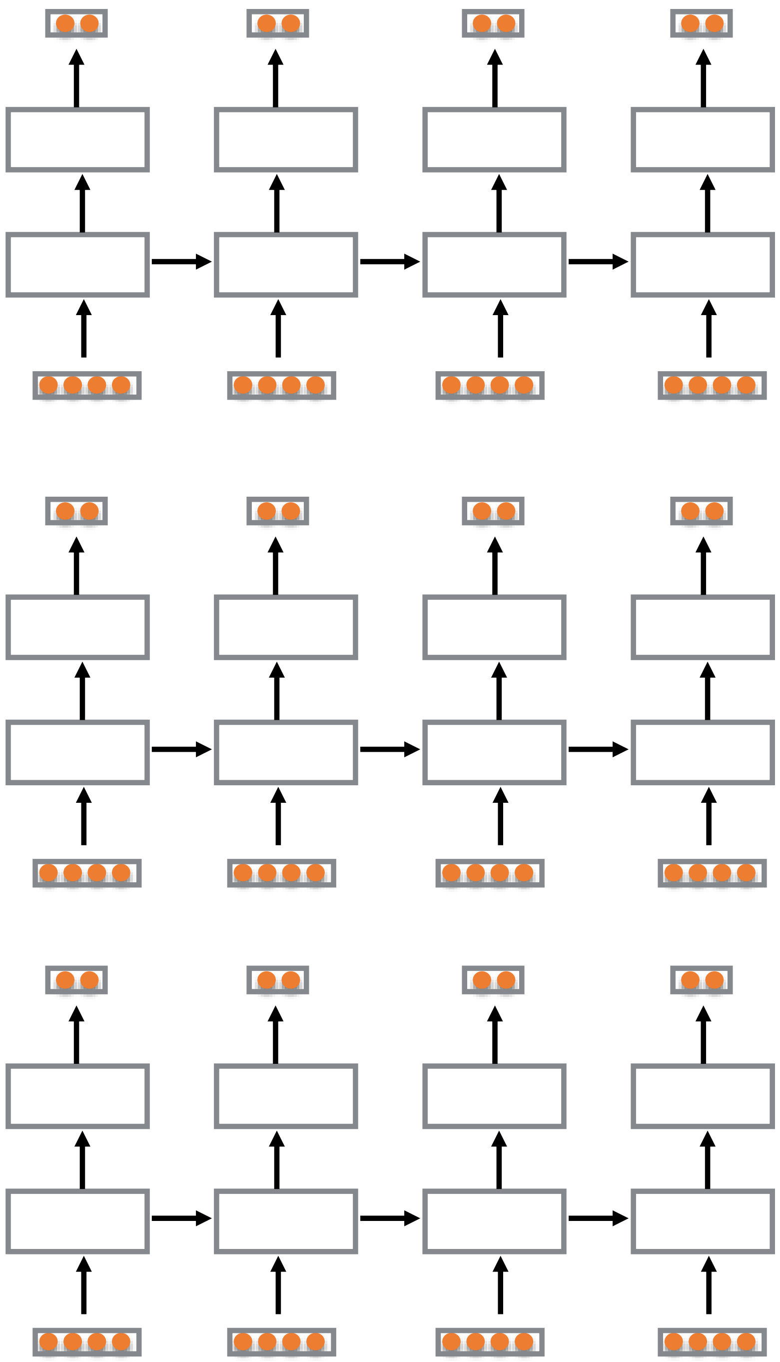
Non-recursive Architectures

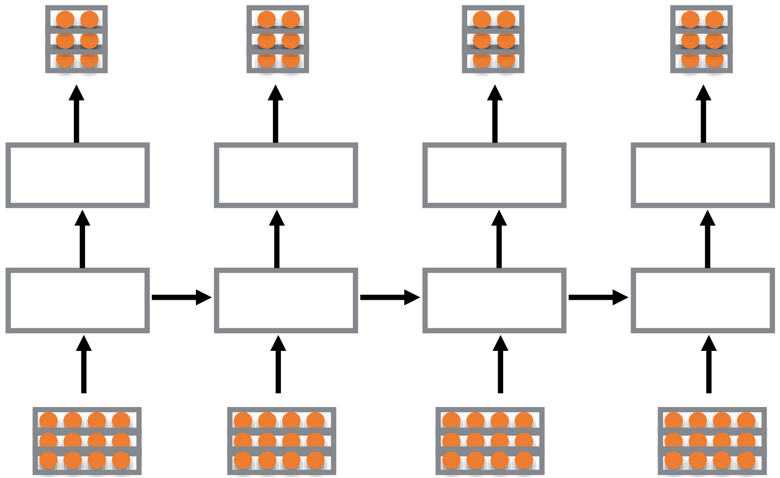
- **Dilated convolutions** for capturing context (Kalchbrenner et al. 2016): single GPU call for entire sentence!



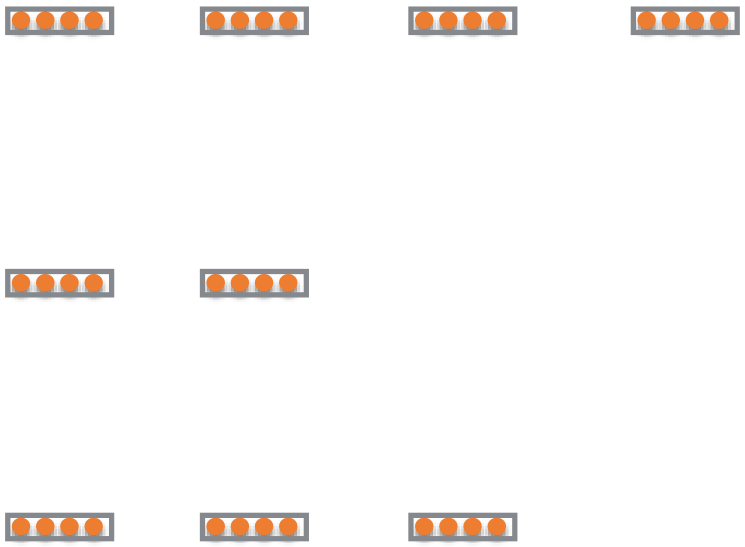
- **Self-attention** that decides which of previous words to focus on (Cheng et al. 2016, Vaswani et al. 2017, covered in detail a few classes): also single GPU call

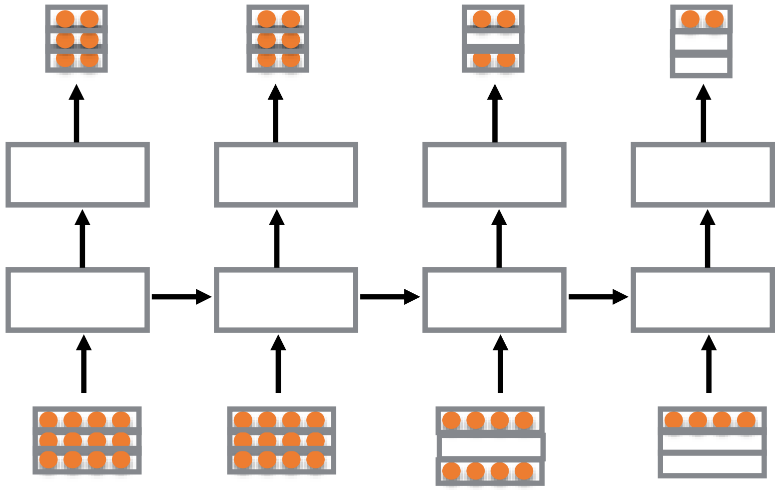


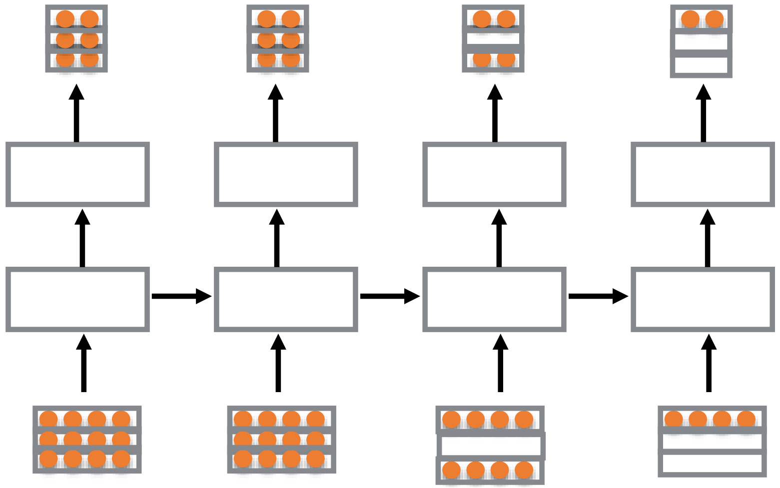




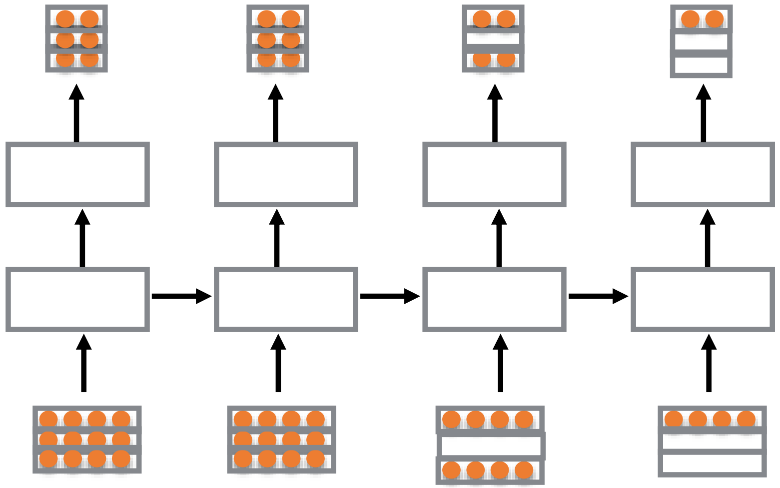
what if the sequences are different lengths?





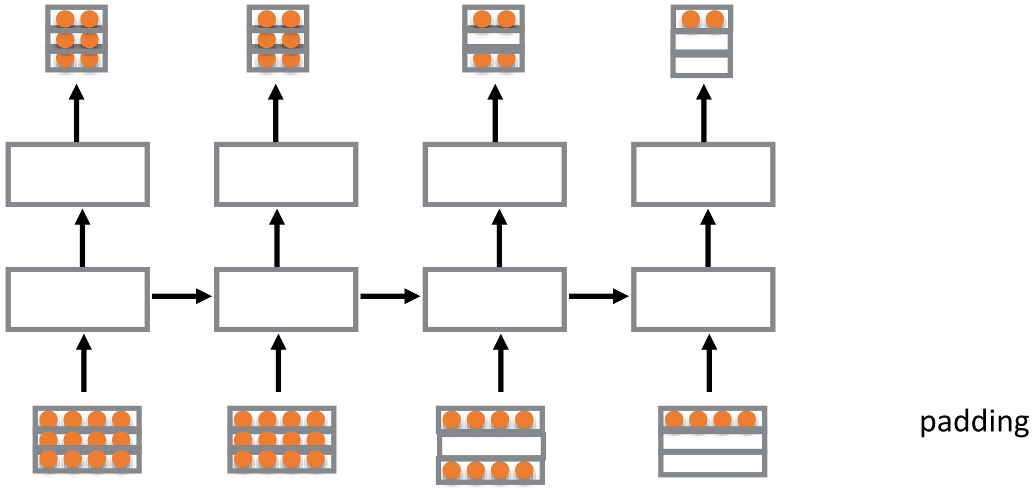


padding



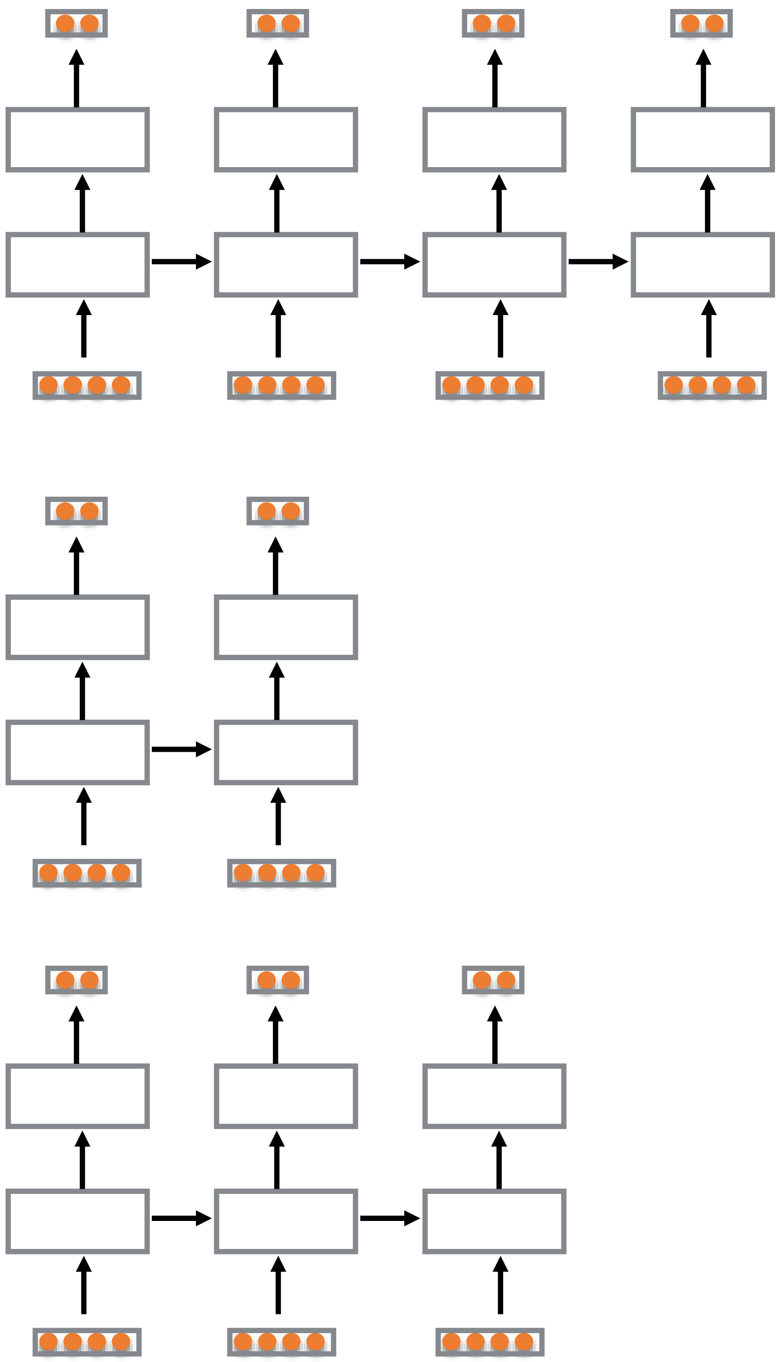
padding

this is how its done in TF, PyTorch.
supported also in DyNet, but...

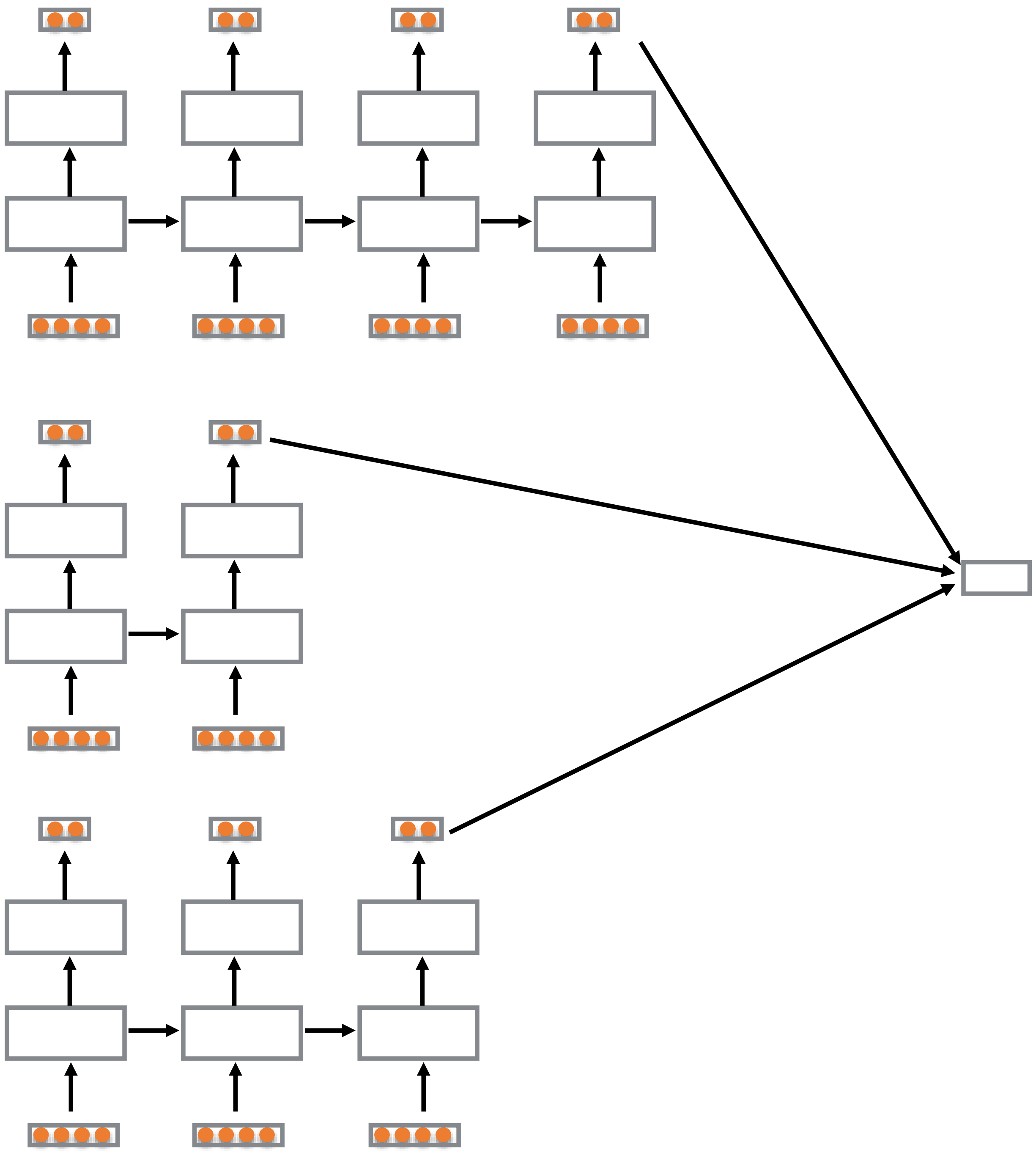


We want better

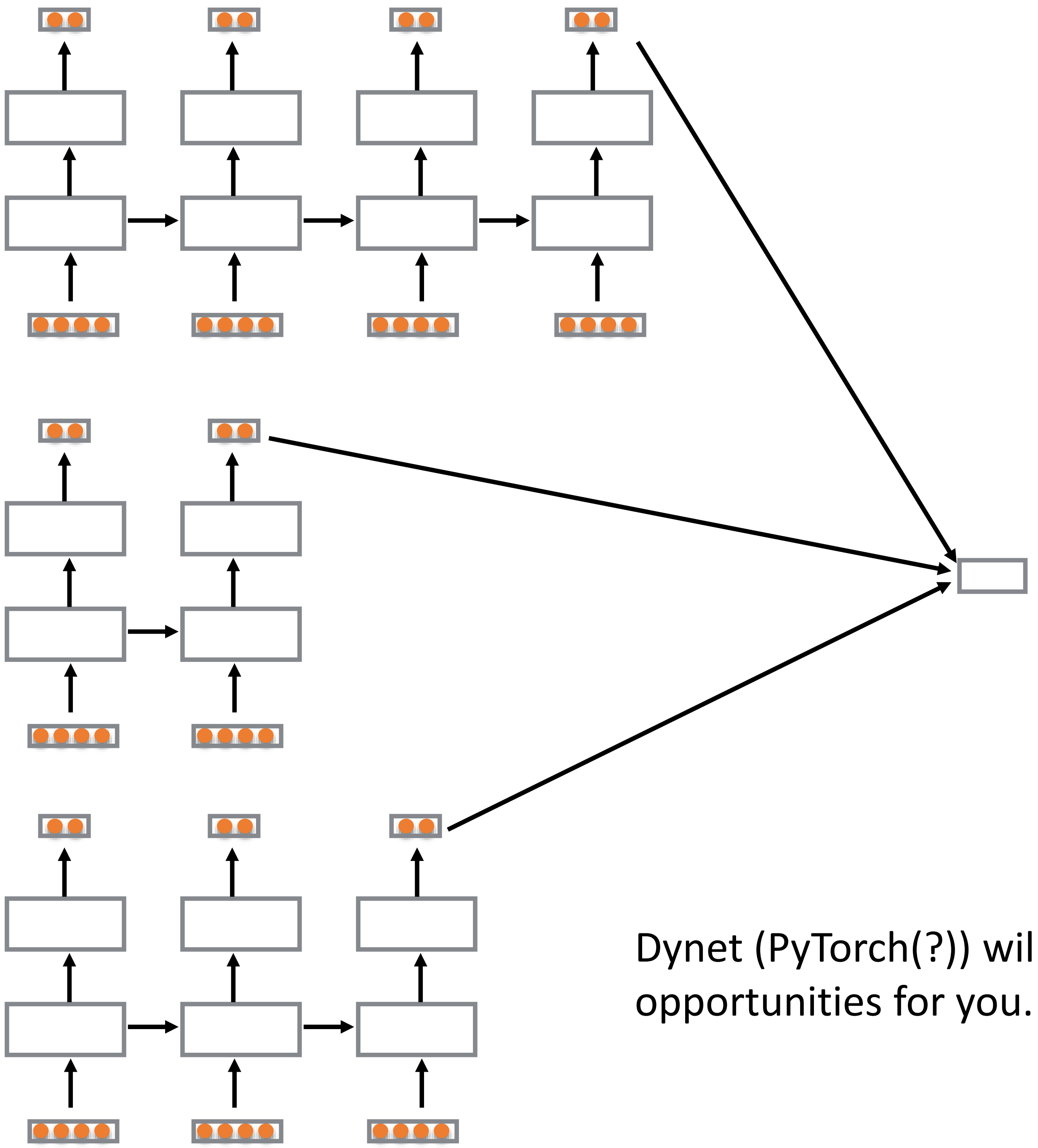
Auto Batching



create a separate
network for each
(easy)

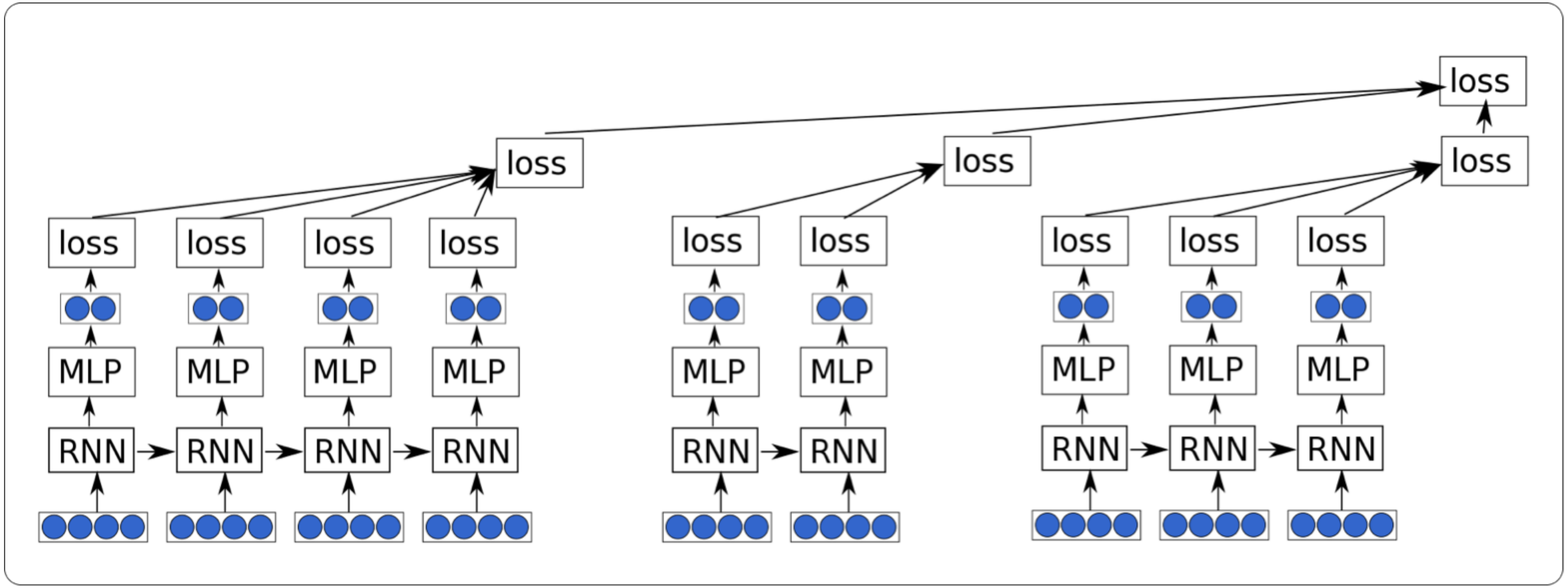


treat them as a single graph



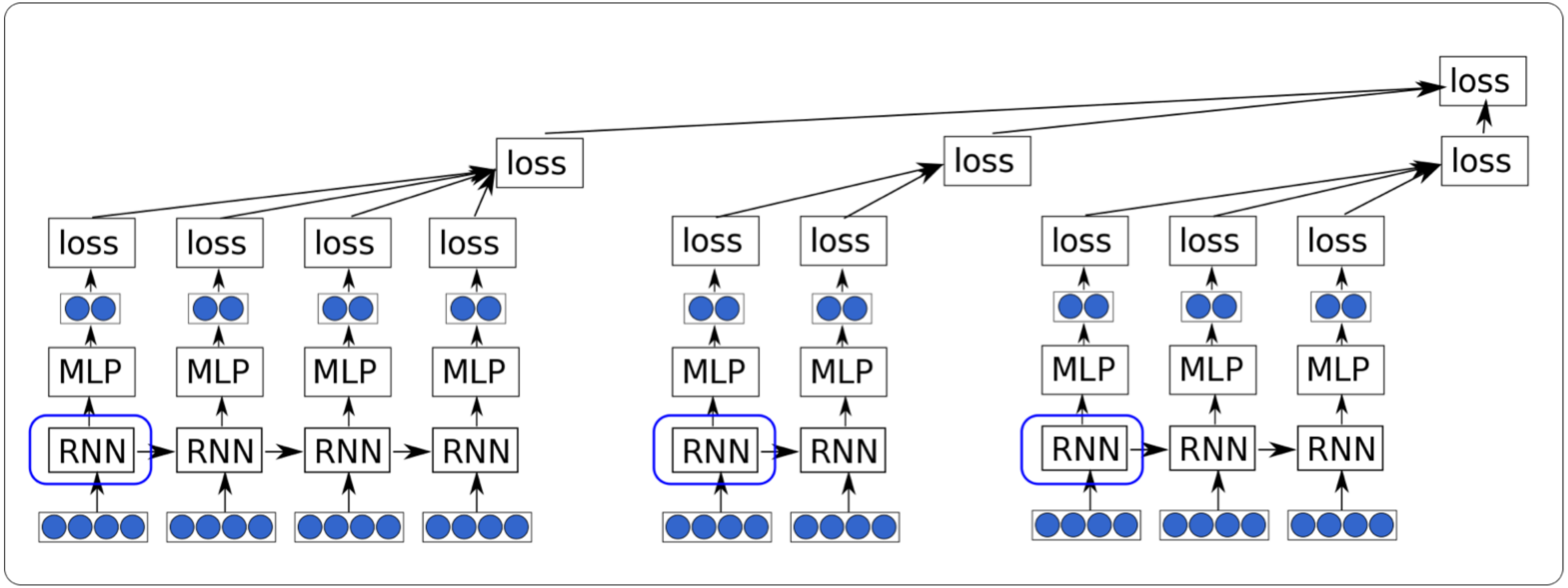
treat them
as a single
graph

Dynet (PyTorch(?)) will identify batching
opportunities for you.

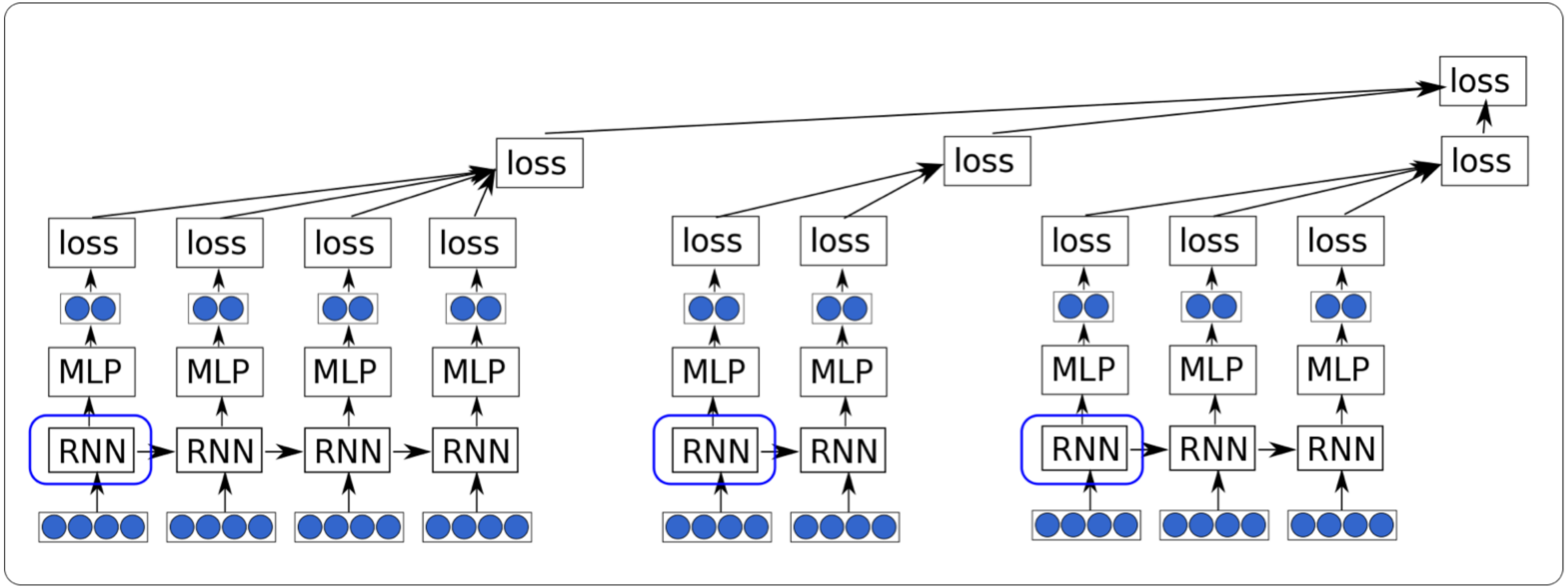


loss
loss
loss
loss
loss
loss
loss
loss
loss
loss
loss
loss
loss
MLP
MLP
MLP
MLP
MLP
MLP
MLP
MLP
MLP
RNN
RNN
RNN
RNN
RNN
RNN
RNN
RNN
RNN

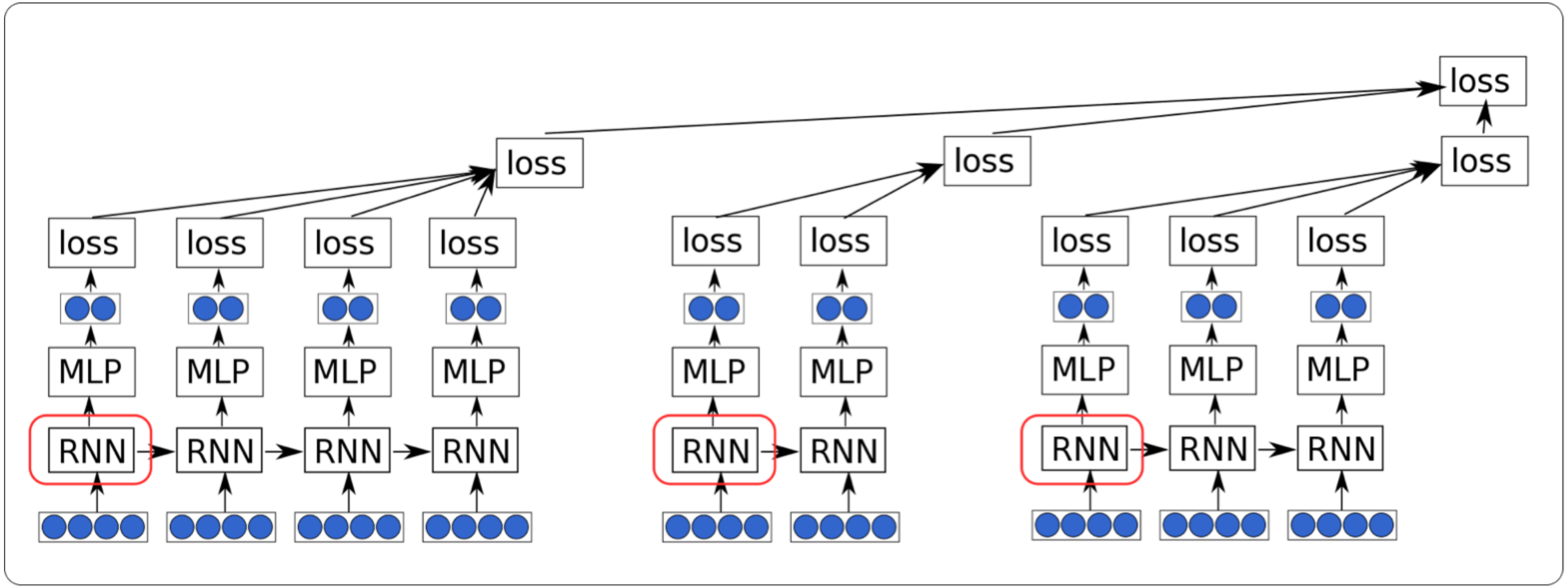
nodes in blue are ready
to be executed


loss
loss
loss
loss
loss
loss
loss
loss
loss
loss
loss
loss
loss
MLP
MLP
MLP
MLP
MLP
MLP
MLP
MLP
MLP
RNN
RNN
RNN
RNN
RNN
RNN
RNN
RNN
RNN

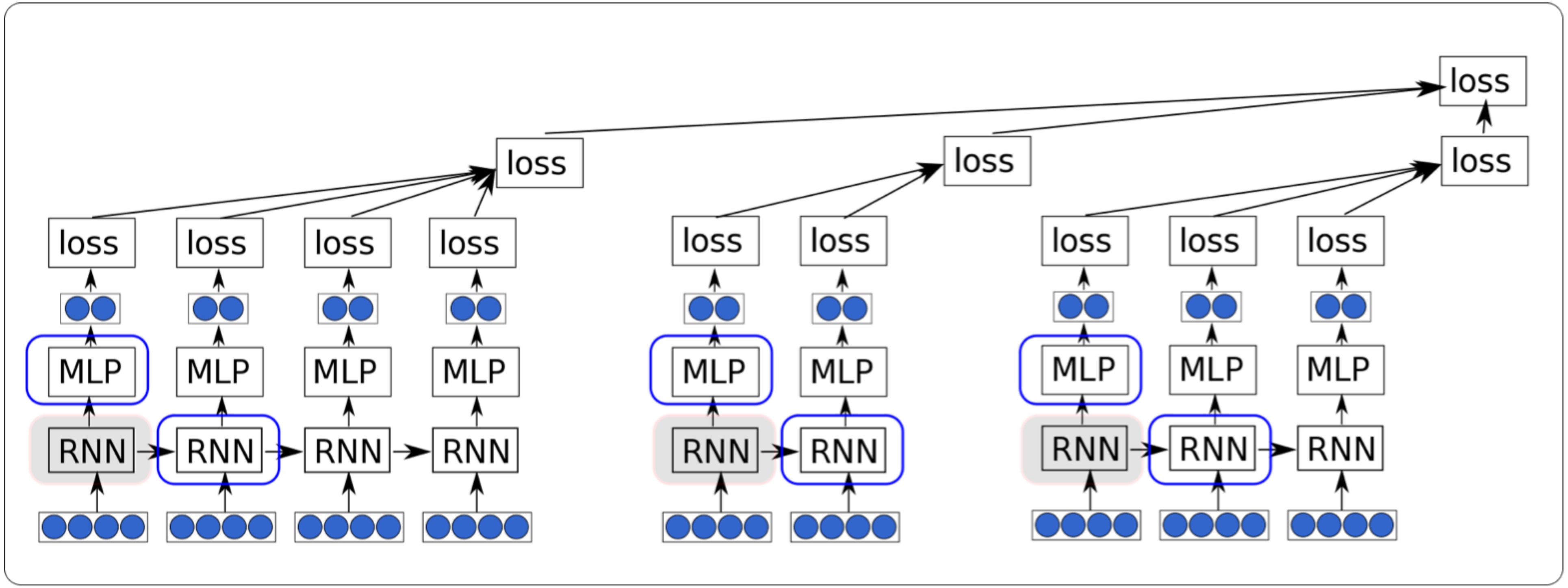
nodes in red will be executed
using batch operations



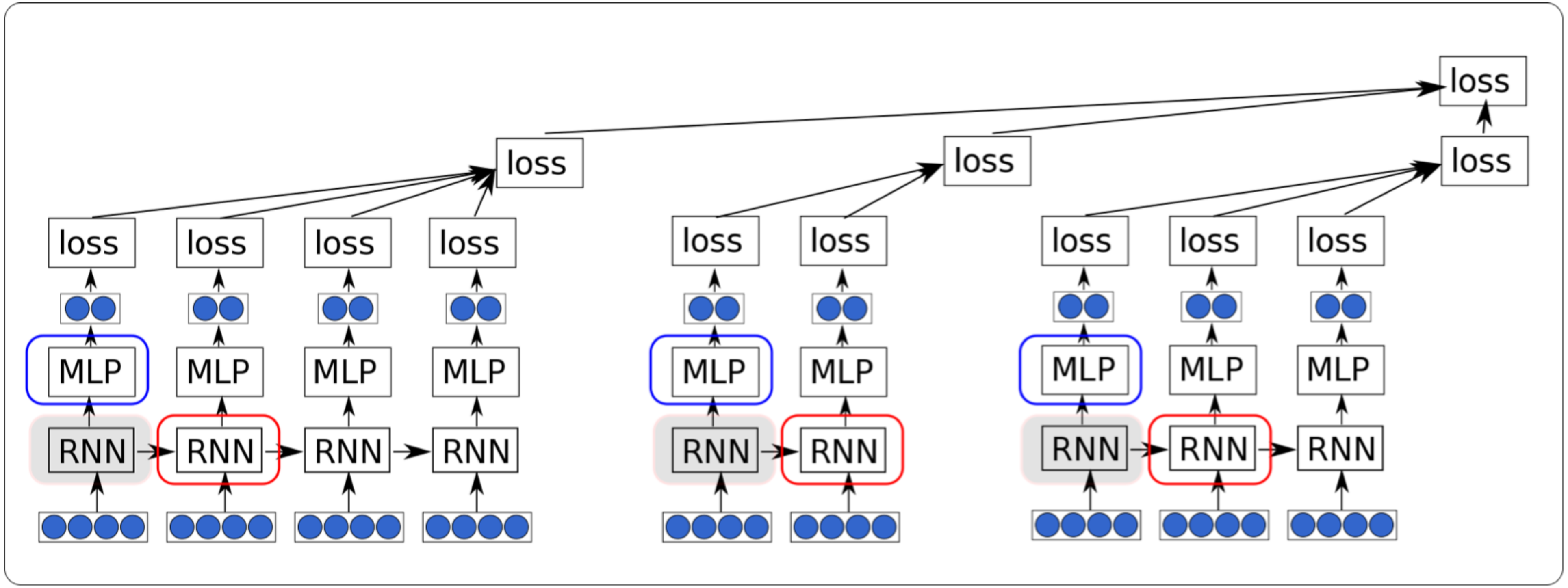
loss
loss
loss
loss
loss loss loss loss
loss loss
loss loss loss
MLP MLP MLP MLP
MLP MLP
MLP MLP MLP
RNN RNN RNN RNN
RNN RNN
RNN RNN RNN



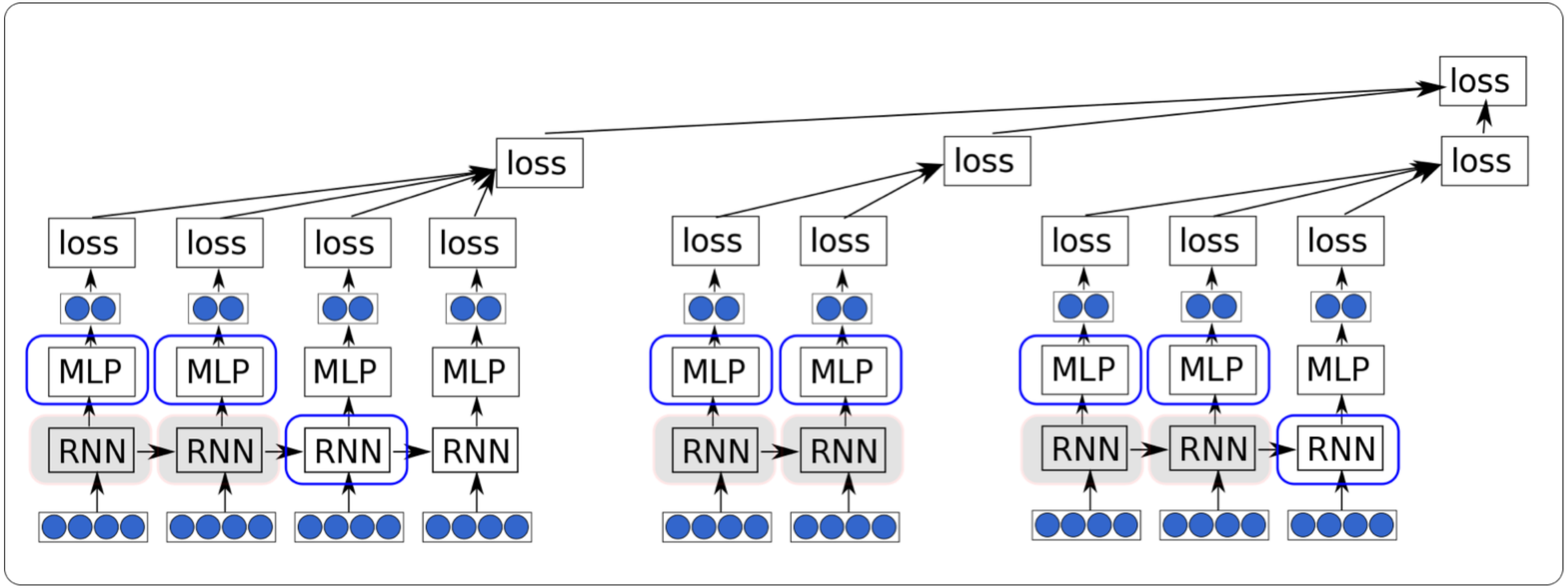
loss
loss
loss
loss
loss
loss
loss
loss
loss
loss
loss
loss
loss
loss
MLP
MLP
MLP
MLP
MLP
MLP
MLP
MLP
MLP
RNN
RNN
RNN
RNN
RNN
RNN
RNN
RNN
RNN



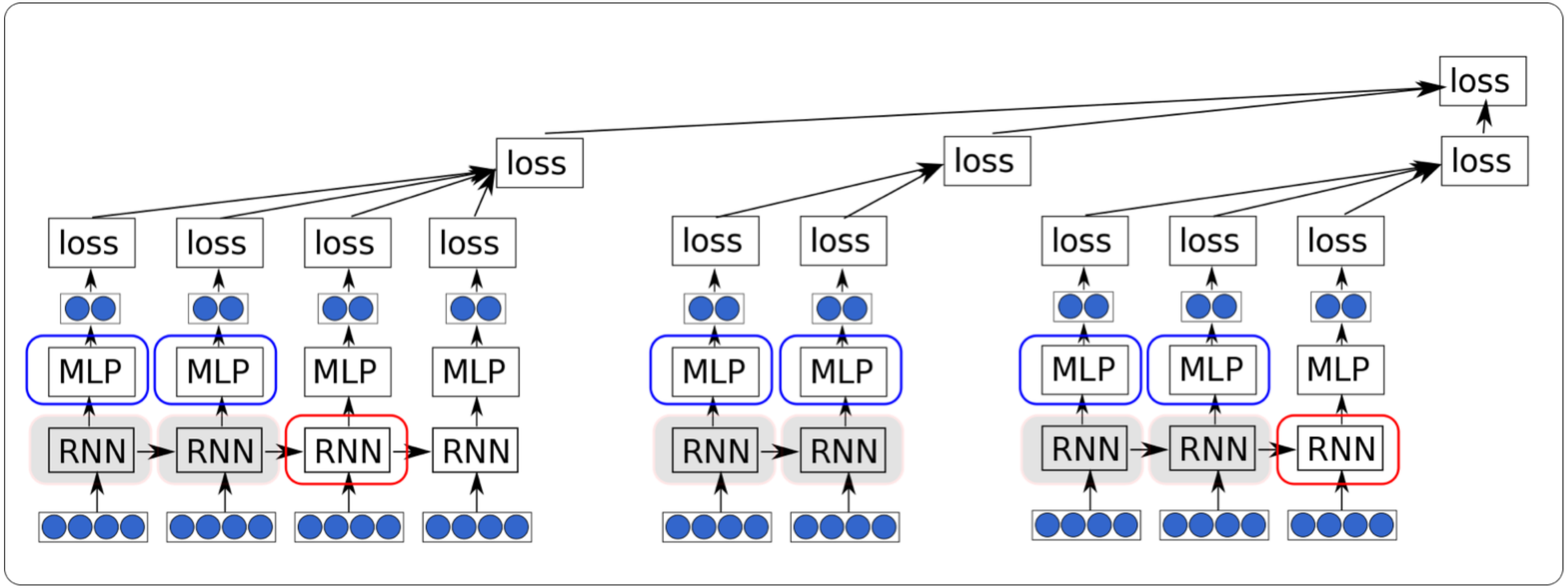
loss
loss
loss
loss
loss
loss
loss
loss
loss
loss
loss
loss
loss
MLP
MLP
MLP
MLP
MLP
MLP
MLP
MLP
MLP
RNN
RNN
RNN
RNN
RNN
RNN
RNN
RNN
RNN



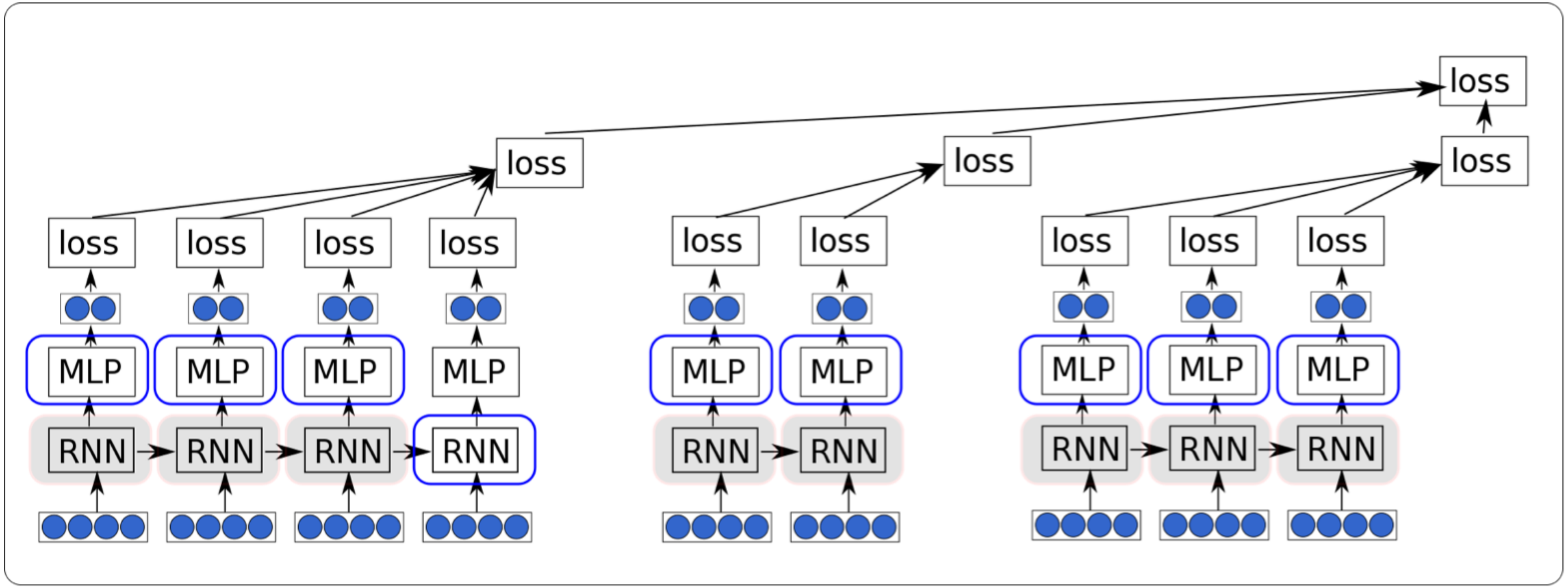
loss
loss
loss
loss
loss
loss
loss
loss
loss
loss
loss
loss
loss
loss
MLP
MLP
MLP
MLP
MLP
MLP
MLP
MLP
MLP
RNN
RNN
RNN
RNN
RNN
RNN
RNN
RNN
RNN

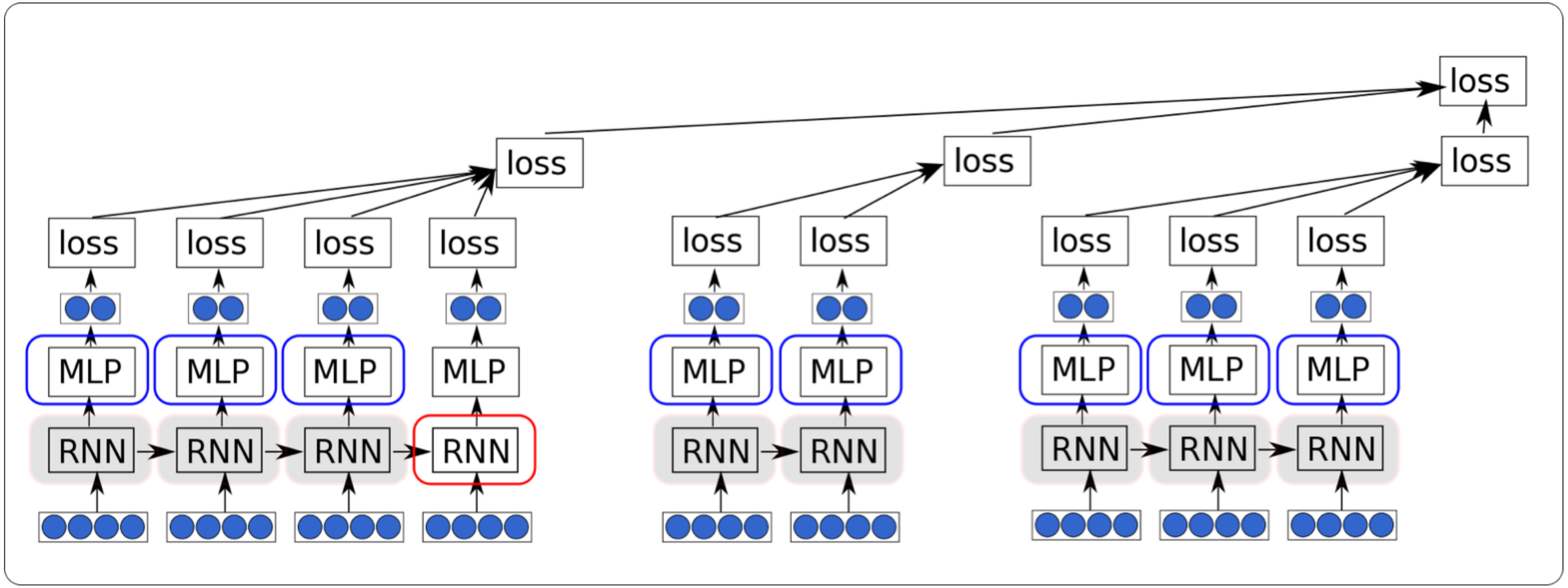


loss
loss
loss
loss
loss
loss
loss
loss
loss
loss
loss
loss
loss
MLP
MLP
MLP
MLP
MLP
MLP
MLP
MLP
MLP
RNN
RNN
RNN
RNN
RNN
RNN
RNN
RNN
RNN

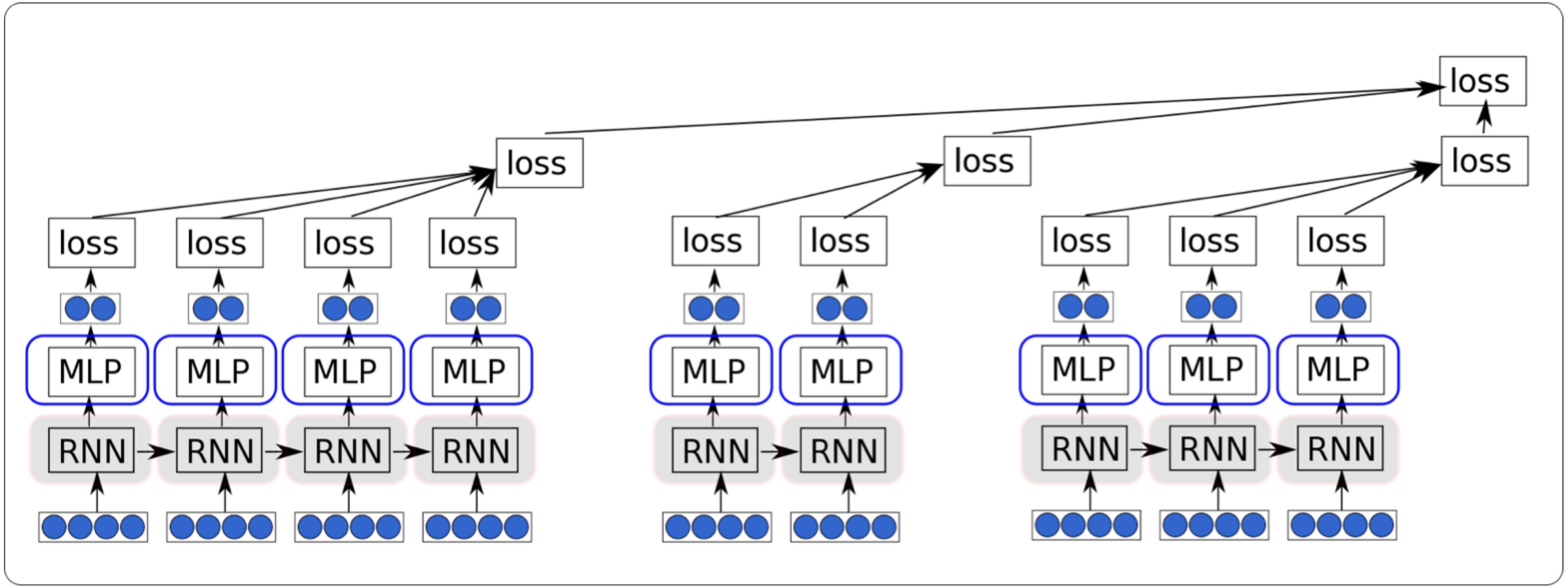


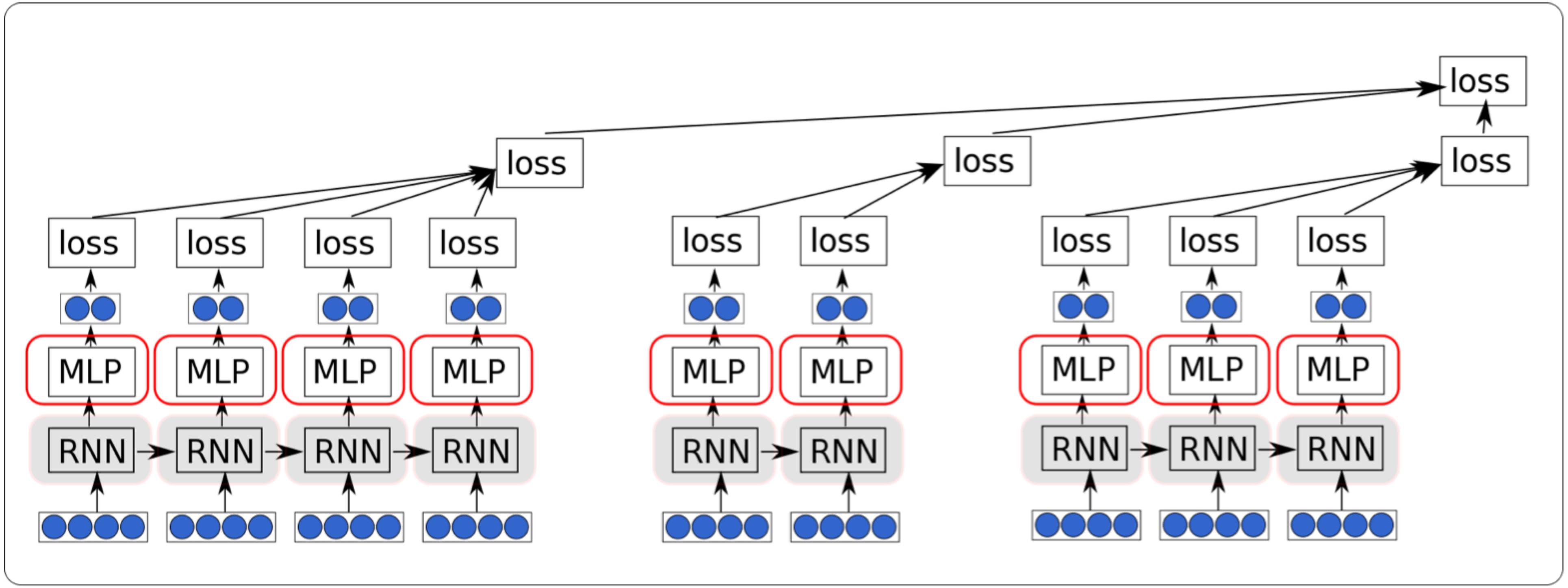
loss
loss
loss
loss
loss
loss
loss
loss
loss
loss
loss
loss
loss
MLP
MLP
MLP
MLP
MLP
MLP
MLP
MLP
MLP
RNN
RNN
RNN
RNN
RNN
RNN
RNN
RNN
RNN

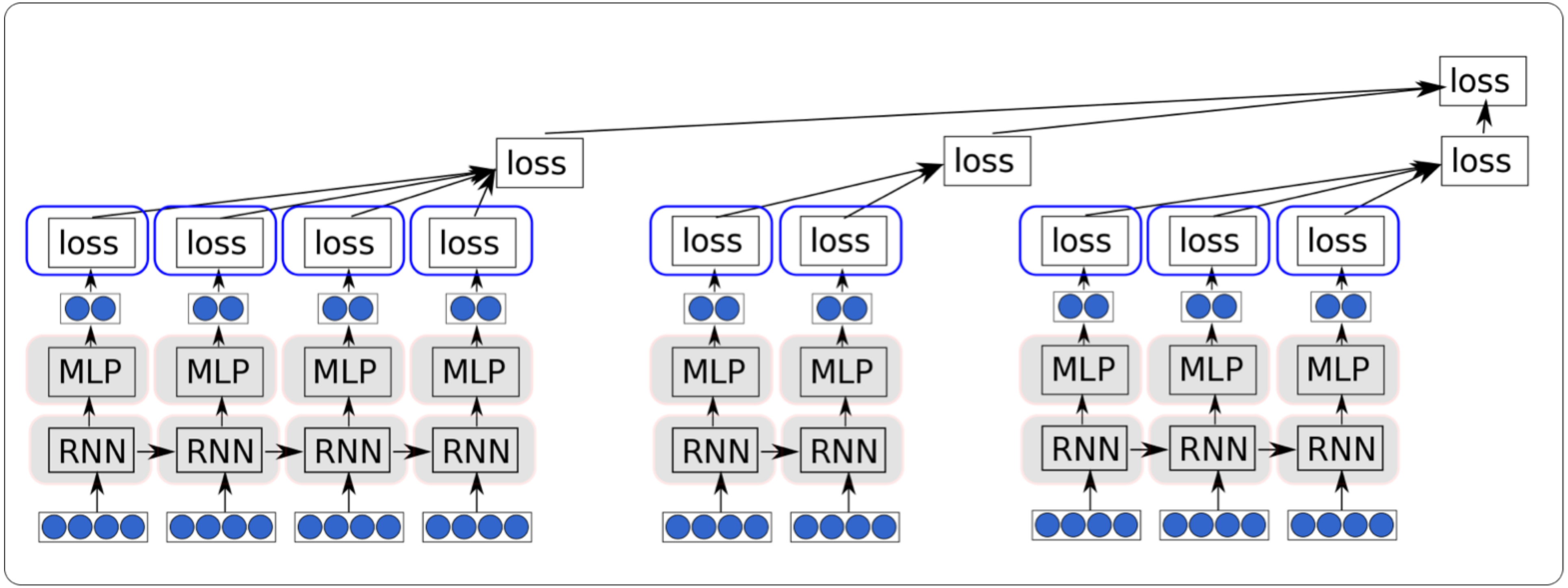

loss
loss
loss
loss
loss
loss
loss
loss
loss
loss
loss
loss
loss
MLP
MLP
MLP
MLP
MLP
MLP
MLP
MLP
MLP
RNN
RNN
RNN
RNN
RNN
RNN
RNN
RNN
RNN

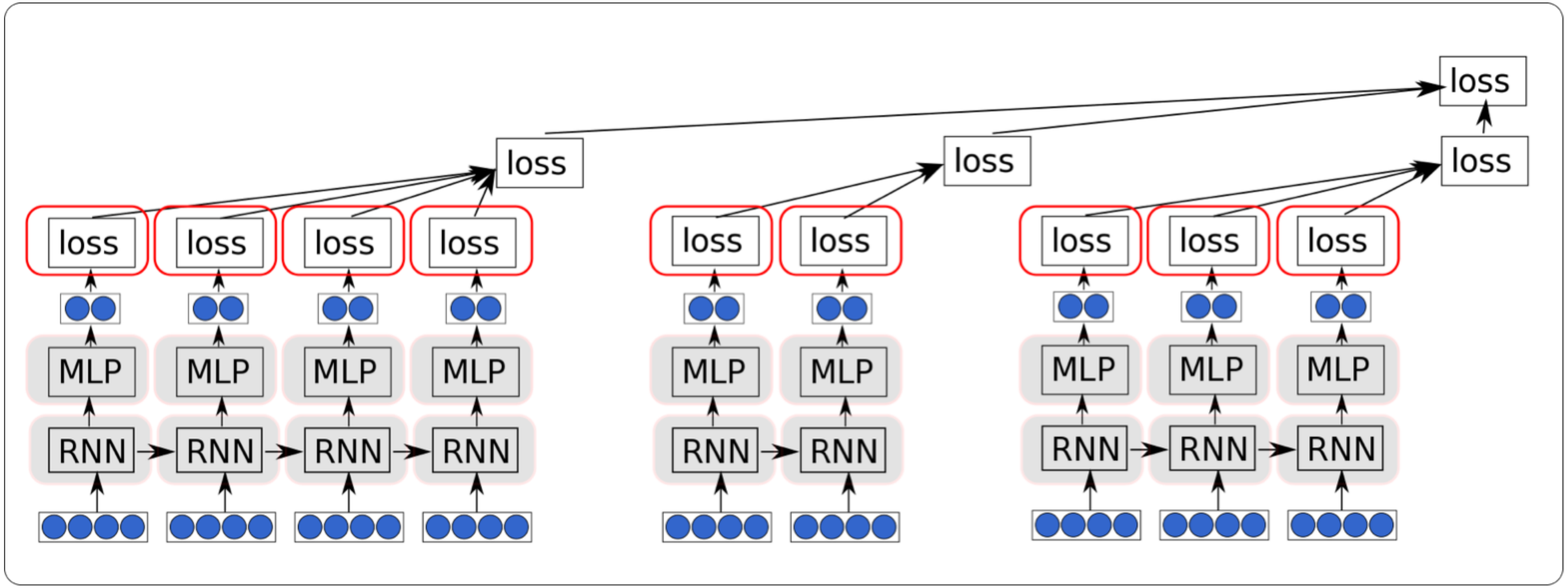


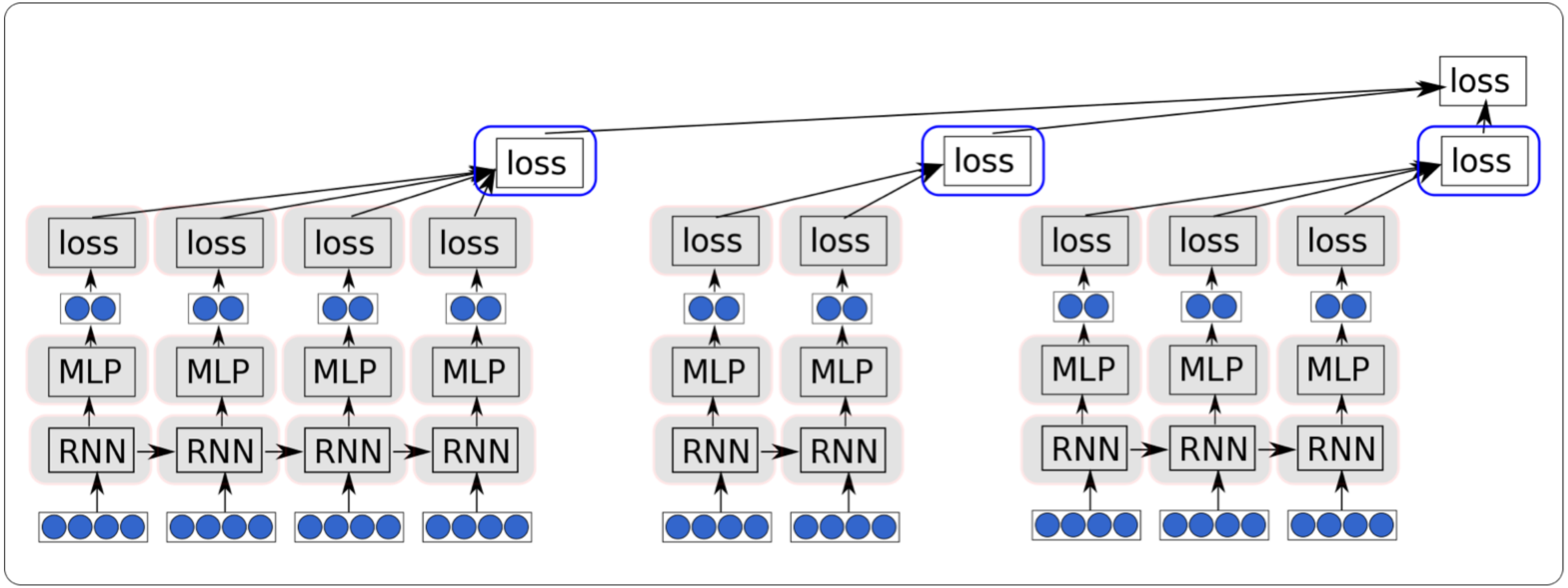
loss
loss
loss
loss
loss
loss
loss
loss
loss
loss
loss
loss
loss
MLP
MLP
MLP
MLP
MLP
MLP
MLP
MLP
MLP
RNN
RNN
RNN
RNN
RNN
RNN
RNN
RNN
RNN

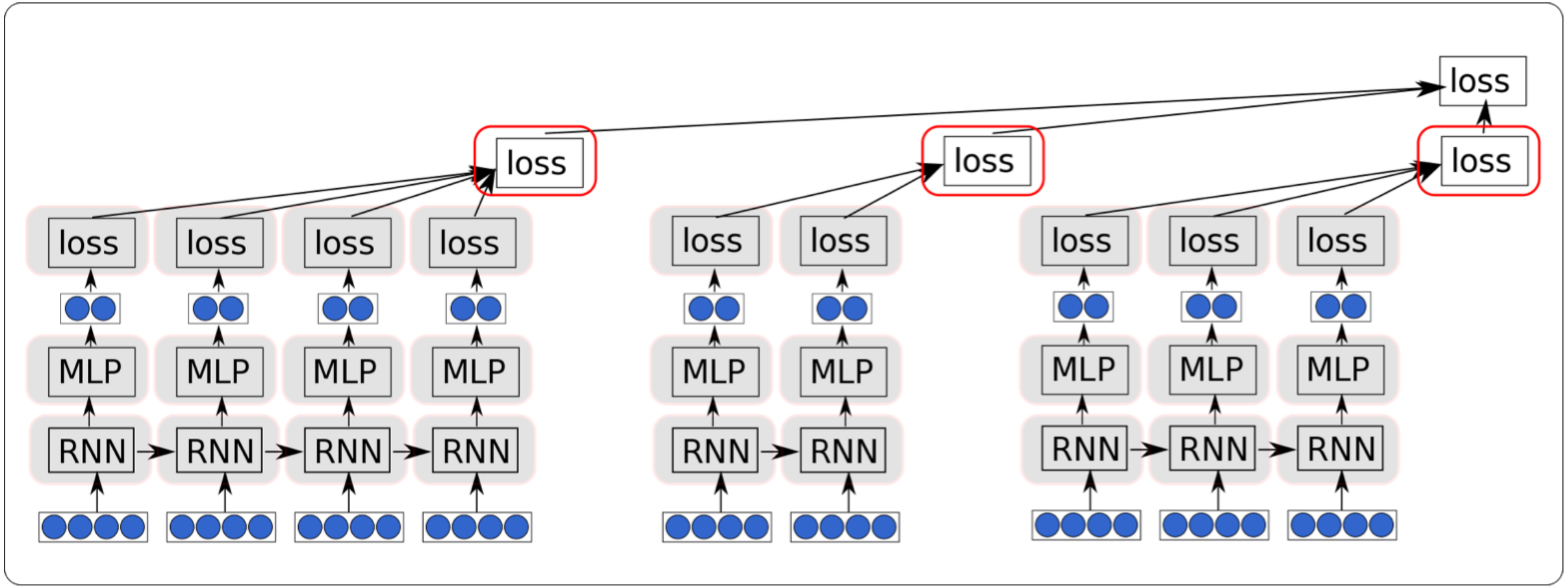

loss
loss
loss
loss
loss
loss
loss
loss
loss
loss
loss
loss
loss
MLP
MLP
MLP
MLP
MLP
MLP
MLP
MLP
MLP
RNN
RNN
RNN
RNN
RNN
RNN
RNN
RNN
RNN


loss
loss
loss
loss
loss
loss
loss
loss
loss
loss
loss
loss
loss
MLP
MLP
MLP
MLP
MLP
MLP
MLP
MLP
MLP
RNN
RNN
RNN
RNN
RNN
RNN
RNN
RNN
RNN

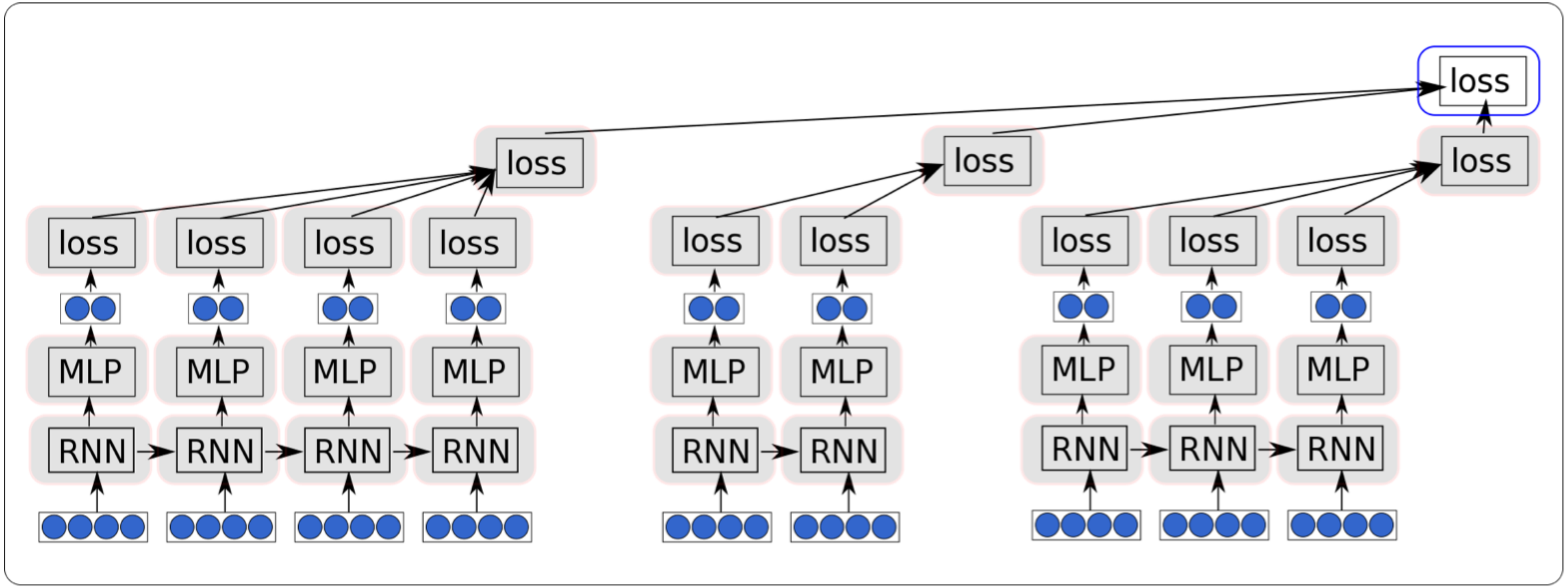

loss
loss
loss
loss
loss
loss
loss
loss
loss
loss
loss
loss
loss
loss
MLP
MLP
MLP
MLP
MLP
MLP
MLP
MLP
MLP
RNN
RNN
RNN
RNN
RNN
RNN
RNN
RNN
RNN

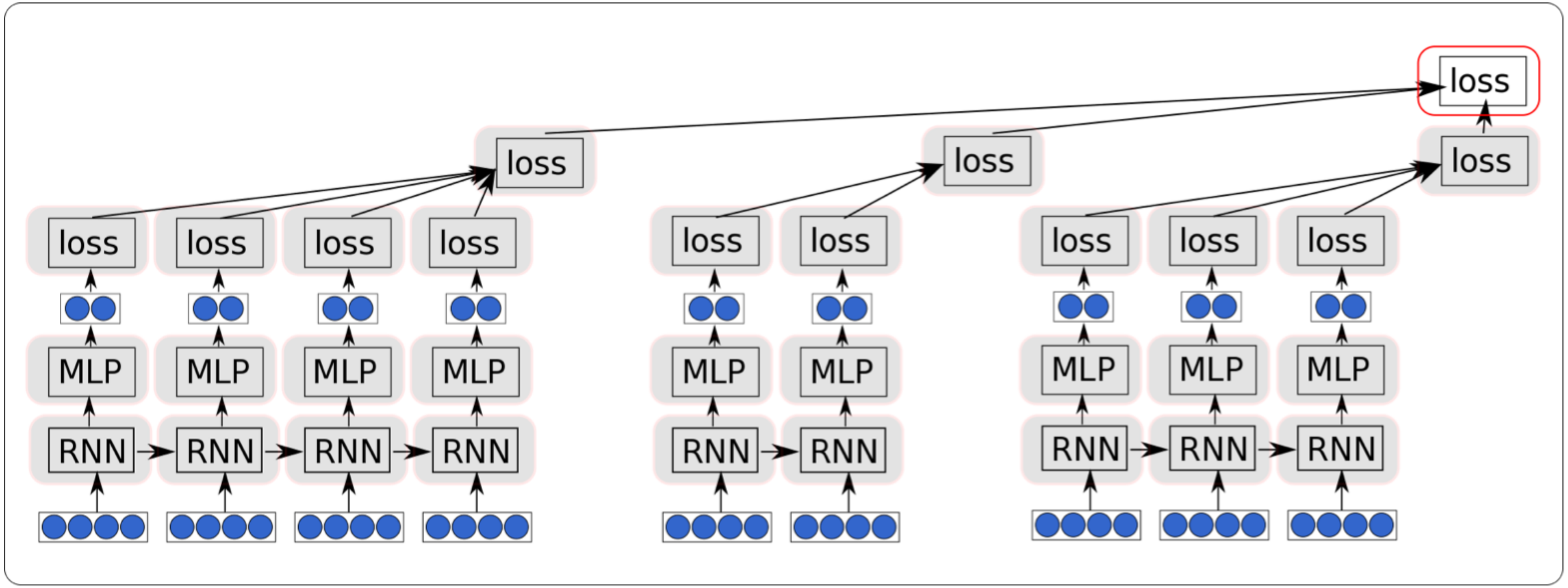

loss
loss
loss
loss
loss
loss
loss
loss
loss
loss
loss
loss
loss
MLP
MLP
MLP
MLP
MLP
MLP
MLP
MLP
MLP
RNN
RNN
RNN
RNN
RNN
RNN
RNN
RNN
RNN


loss
loss
loss
loss
loss
loss
loss
loss
loss
loss
loss
loss
loss
MLP
MLP
MLP
MLP
MLP
MLP
MLP
MLP
MLP
RNN
RNN
RNN
RNN
RNN
RNN
RNN
RNN
RNN

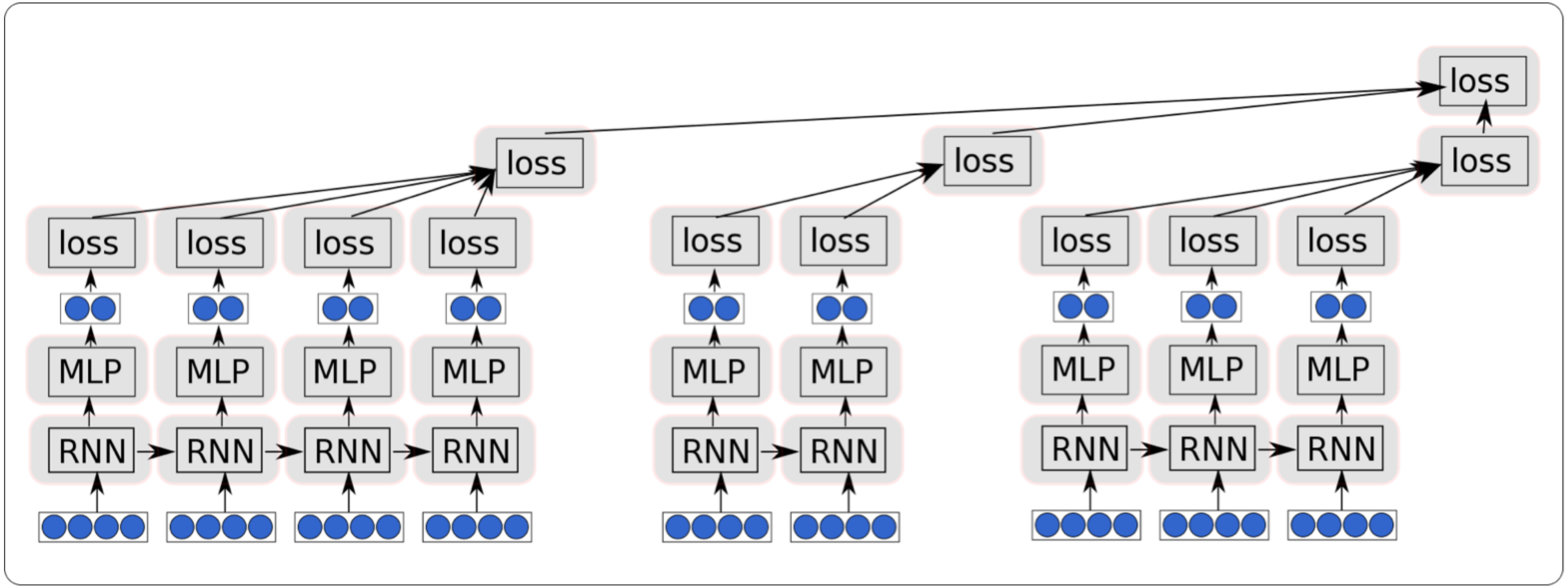


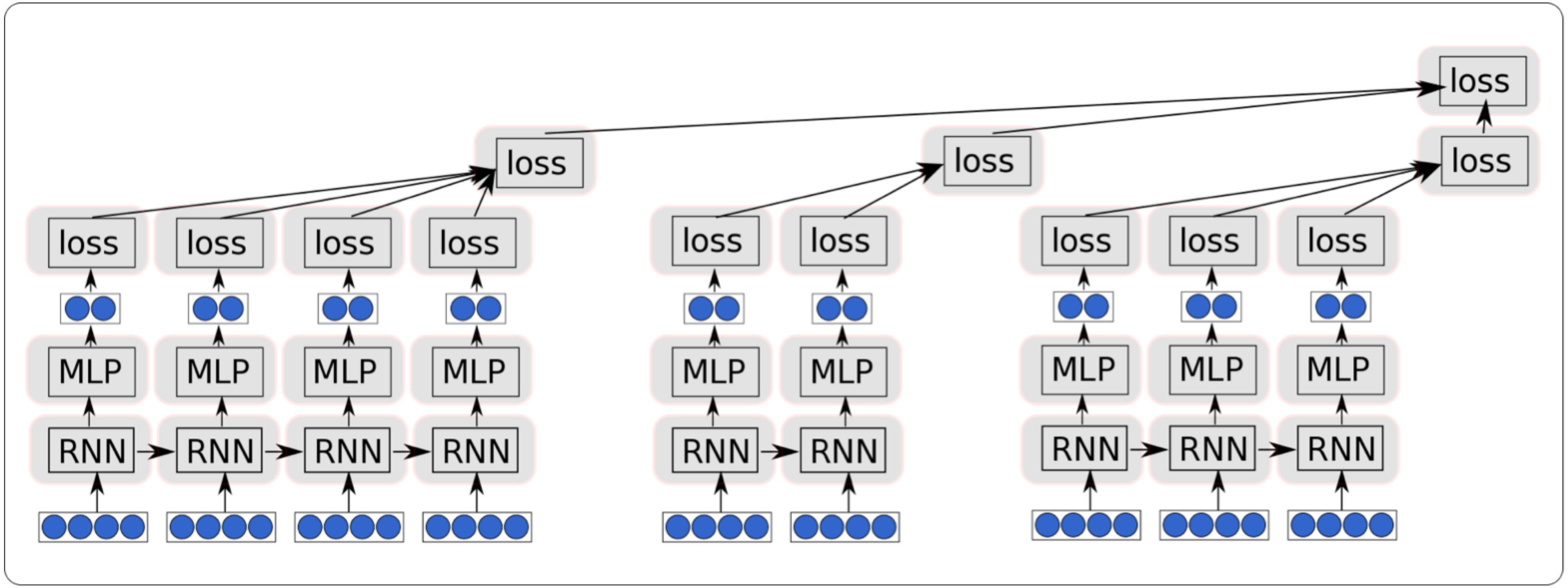
loss
loss
loss
loss
loss
loss
loss
loss
loss
loss
loss
loss
loss
MLP
MLP
MLP
MLP
MLP
MLP
MLP
MLP
MLP
RNN
RNN
RNN
RNN
RNN
RNN
RNN
RNN
RNN


loss
loss
loss
loss
loss loss loss loss
loss loss
loss loss loss
MLP MLP MLP MLP
MLP MLP
MLP MLP MLP
RNN RNN RNN RNN
RNN RNN
RNN RNN RNN



loss
loss
loss
loss
loss
loss
loss
loss
loss
loss
loss
loss
loss
MLP
MLP
MLP
MLP
MLP
MLP
MLP
MLP
MLP
RNN
RNN
RNN
RNN
RNN
RNN
RNN
RNN
RNN


loss
loss
loss
loss
loss
loss
loss
loss
loss
loss
loss
loss
loss
MLP
MLP
MLP
MLP
MLP
MLP
MLP
MLP
MLP
RNN
RNN
RNN
RNN
RNN
RNN
RNN
RNN
RNN



<u>note</u>: batching operations, not inputs.

Efficiency Considerations when Implementing an LSTM

$$
\begin{aligned}
R_{LSTM}(\mathbf{s_{j-1}}, \mathbf{x_j}) =& [\mathbf{c_j}; \mathbf{h_j}] \\
\mathbf{c_j} =& \mathbf{c_{j-1}} \odot \mathbf{f} + \mathbf{g} \odot \mathbf{i} \\
\mathbf{h_j} =& \tanh(\mathbf{c_j}) \odot \mathbf{o} \\
\mathbf{i} =& \sigma(\mathbf{W^{xi}} \cdot \mathbf{x_j} + \mathbf{W^{hi}} \cdot \mathbf{h_{j-1}}) \\
\mathbf{f} =& \sigma(\mathbf{W^{xf}} \cdot \mathbf{x_j} + \mathbf{W^{hf}} \cdot \mathbf{h_{j-1}}) \\
\mathbf{o} =& \sigma(\mathbf{W^{xo}} \cdot \mathbf{x_j} + \mathbf{W^{ho}} \cdot \mathbf{h_{j-1}}) \\
\mathbf{g} =& \tanh(\mathbf{W^{xg}} \cdot \mathbf{x_j} + \mathbf{W^{hg}} \cdot \mathbf{h_{j-1}})
\end{aligned}
$$

Efficiency Considerations when Implementing an LSTM

$$
\begin{aligned}
R_{LSTM}(\mathbf{s_{j-1}}, \mathbf{x_j}) =& [\mathbf{c_j}; \mathbf{h_j}] \\
\mathbf{c_j} =& \mathbf{c_{j-1}} \odot \mathbf{f} + \mathbf{g} \odot \mathbf{i} \\
\mathbf{h_j} =& \tanh(\mathbf{c_j}) \odot \mathbf{o} \\
\mathbf{i} =& \sigma(\mathbf{W^{xi}} \cdot \mathbf{x_j} + \mathbf{W^{hi}} \cdot \mathbf{h_{j-1}}) \\
\mathbf{f} =& \sigma(\mathbf{W^{xf}} \cdot \mathbf{x_j} + \mathbf{W^{hf}} \cdot \mathbf{h_{j-1}}) \\
\mathbf{o} =& \sigma(\mathbf{W^{xo}} \cdot \mathbf{x_j} + \mathbf{W^{ho}} \cdot \mathbf{h_{j-1}}) \\
\mathbf{g} =& \tanh(\mathbf{W^{xg}} \cdot \mathbf{x_j} + \mathbf{W^{hg}} \cdot \mathbf{h_{j-1}})
\end{aligned}
$$

all gates computations can be done in single mat-mat op.

Speed Trick 1:
Don’t Repeat Operations

- Something that you can do once at the beginning of the sentence, don’t do it for every word!

Bad

```
for x in words_in_sentence:
  vals.append( W * c + x )
```

Speed Trick 1:
Don’t Repeat Operations

- Something that you can do once at the beginning of the sentence, don’t do it for every word!

Bad

```
for x in words_in_sentence:
  vals.append( W * c + x )
```

Good

```
W_c = W * c
for x in words_in_sentence:
  vals.append( W_c + x )
```

Speed Trick 2: Reduce # of Operations

- e.g. can you combine multiple matrix-vector multiplies into a single matrix-matrix multiply? Do so!

Bad

```
for x in words_in_sentence:
  vals.append( W * x )
val = dy.concatenate(vals)
```

Speed Trick 2: Reduce # of Operations

- e.g. can you combine multiple matrix-vector multiplies into a single matrix-matrix multiply? Do so!

Bad

```
for x in words_in_sentence:
  vals.append( W * x )
val = dy.concatenate(vals)
```

Good

```
X = dy.concatenate_cols(words_in_sentence)
val = W * X
```

Speed Trick 3: Reduce CPU-GPU Data Movement

- Try to **avoid memory moves** between CPU and GPU.

- When you do move memory, try to do it as early as possible (GPU operations are asynchronous)

Bad

```
for x in words_in_sentence:
  # input data for x
  # do processing
```

Speed Trick 3:
Reduce CPU-GPU Data Movement

- Try to **avoid memory moves** between CPU and GPU.
- When you do move memory, try to do it as early as possible (GPU operations are asynchronous)

Bad

```
for x in words_in_sentence:
  # input data for x
  # do processing
```

Good

```
# input data for whole sentence
for x in words_in_sentence:
  # do processing
```