Applied Parallel Computing
parallel-computing.pro

# TensorFlow Framework

# MNIST Skeleton

```
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

# Imports
import numpy as np
import tensorflow as tf

from tensorflow.contrib import learn
from tensorflow.contrib.learn.python.learn.estimators import model_fn as model_fn_lib

tf.logging.set_verbosity(tf.logging.INFO)

# Our application logic will be added here

if __name__ == "__main__":
  tf.app.run()
```

# MNIST Layers

- Let's build a model to classify the images in the MNIST dataset using the following CNN architecture:
  - **Convolutional Layer #1**: Applies 32 5x5 filters (extracting 5x5-pixel subregions), with ReLU activation function
  - **Pooling Layer #1**: Performs max pooling with a 2x2 filter and stride of 2 (which specifies that pooled regions do not overlap)
  - **Convolutional Layer #2**: Applies 64 5x5 filters, with ReLU activation function
  - **Pooling Layer #2**: Again, performs max pooling with a 2x2 filter and stride of 2
  - **Dense Layer #1**: 1,024 neurons, with dropout regularization rate of 0.4 (probability of 0.4 that any given element will be dropped during training)
  - **Dense Layer #2 (Logits Layer)**: 10 neurons, one for each digit target class (0–9).

# tf.layers

- The tf.layers module contains methods to create each of the three layer types above:
  - conv2d(). Constructs a two-dimensional convolutional layer. Takes number of filters, filter kernel size, padding, and activation function as arguments.
  - max_pooling2d(). Constructs a two-dimensional pooling layer using the max-pooling algorithm. Takes pooling filter size and stride as arguments.
  - dense(). Constructs a dense layer. Takes number of neurons and activation function as arguments.

# MNIST

- Open cnn_mnist.py and add the following cnn_model_fn function, which conforms to the interface expected by TensorFlow's Estimator API.
  - cnn_mnist.py takes MNIST feature data, labels, and model mode (TRAIN, EVAL, INFER) as arguments;
  - configures the CNN;
  - returns predictions, loss, and a training operation

# MNIST layers

```
def cnn_model_fn(features, labels, mode):
  """Model function for CNN."""
  # Input Layer
  input_layer = tf.reshape(features, [-1, 28, 28, 1])

  # Convolutional Layer #1
  conv1 = tf.layers.conv2d(
      inputs=input_layer,
      filters=32,
      kernel_size=[5, 5],
      padding="same",
      activation=tf.nn.relu)

  # Pooling Layer #1
  pool1 = tf.layers.max_pooling2d(inputs=conv1, pool_size=[2, 2], strides=2)
```

# MNIST layers

```
# Convolutional Layer #2 and Pooling Layer #2
  conv2 = tf.layers.conv2d(
      inputs=pool1,
      filters=64,
      kernel_size=[5, 5],
      padding="same",
      activation=tf.nn.relu)
  pool2 = tf.layers.max_pooling2d(inputs=conv2, pool_size=[2, 2], strides=2)

  # Dense Layer
  pool2_flat = tf.reshape(pool2, [-1, 7 * 7 * 64])
  dense = tf.layers.dense(inputs=pool2_flat, units=1024, activation=tf.nn.relu)
  dropout = tf.layers.dropout(
      inputs=dense, rate=0.4, training=mode == learn.ModeKeys.TRAIN)

  # Logits Layer
  logits = tf.layers.dense(inputs=dropout, units=10)
```

# MNIST layers

```
# Calculate Loss (for both TRAIN and EVAL modes)
if mode != learn.ModeKeys.INFER:
  onehot_labels = tf.one_hot(indices=tf.cast(labels, tf.int32), depth=10)
  loss = tf.losses.softmax_cross_entropy(
      onehot_labels=onehot_labels, logits=logits)

# Configure the Training Op (for TRAIN mode)
if mode == learn.ModeKeys.TRAIN:
  train_op = tf.contrib.layers.optimize_loss(
      loss=loss,
      global_step=tf.contrib.framework.get_global_step(),
      learning_rate=0.001,
      optimizer="SGD")
```

# MNIST layers

```
# Generate Predictions
  predictions = {
      "classes": tf.argmax(
          input=logits, axis=1),
      "probabilities": tf.nn.softmax(
          logits, name="softmax_tensor")
  }

  # Return a ModelFnOps object
  return model_fn_lib.ModelFnOps(
      mode=mode, predictions=predictions, loss=loss, train_op=train_op)
```

# Input layer

- Parameters are:
  - batch_size. Size of the subset of examples to use when performing gradient descent during training.
  - image_width. Width of the example images.
  - image_height. Height of the example images.
  - channels. Number of color channels in the example images. For color images, the number of channels is 3 (red, green, blue). For monochrome images, there is just 1 channel (black).

# Convolutional layer

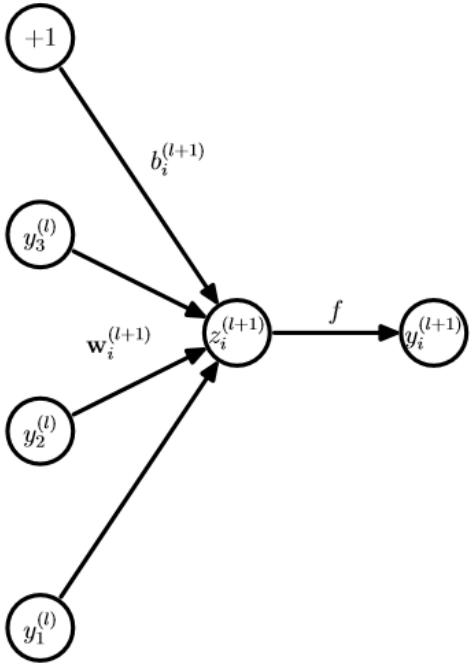
- Parameters are:
  - The inputs argument specifies our input tensor, which must have the shape [batch_size, image_width, image_height, channels].
  - The filters argument specifies the number of filters to apply and kernel_size.
  - The padding argument specifies one of two enumerated values: valid (default value) or same.
    - ✓ (Without padding, a 5x5 convolution over a 28x28 tensor will produce a 24x24 tensor, as there are 24x24 locations to extract a 5x5 tile from a 28x28 grid.)
  - The activation argument specifies the activation function to apply to the output of the convolution.

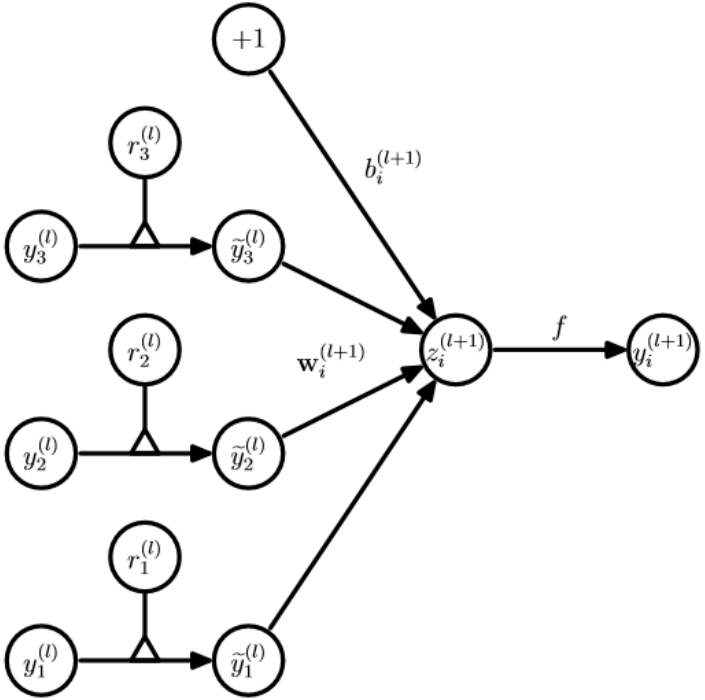
# Pooling layer

- Parameters are:
  - The pool_size specifies the size of the max pooling filter as [width, height]
  - The strides specifies the size of the stride
- Our output tensor produced by max_pooling2d() (pool1) has a shape of [batch_size, 14, 14, 1]: the 2x2 filter reduces width and height by 50%.

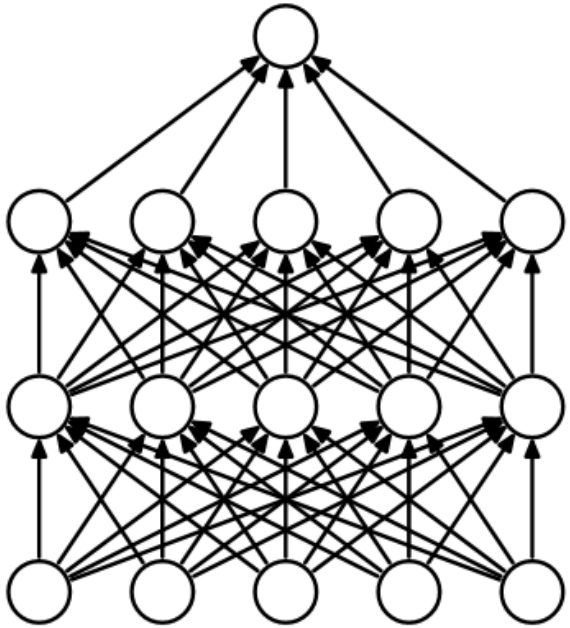
- Parameters are:
  - The inputs: our flattened feature map, pool2_flat.
    - ✓ The units argument specifies the number of neurons in the dense layer (1,024).
    - ✓ The activation argument takes the activation function.
  - To help improve the results of our model, we also apply dropout regularization to our dense layer, using the dropout method in layers:
    - ✓ dropout = tf.layers.dropout(inputs=dense, rate=0.4, training=mode == learn.ModeKeys.TRAIN)

# Dropout

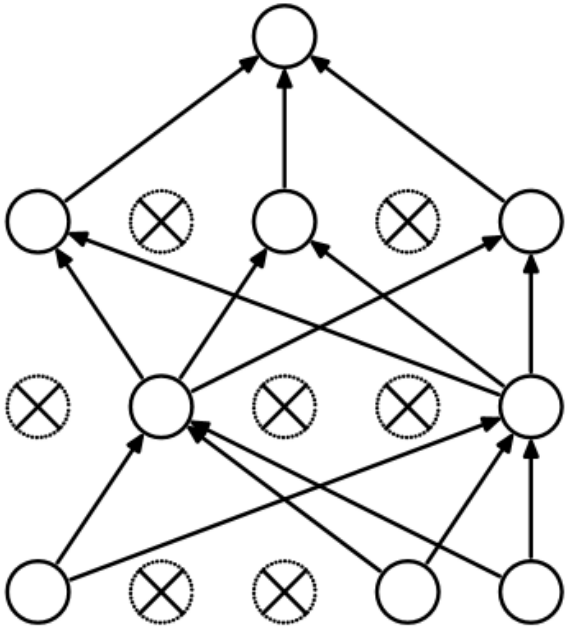

+1
$b_i^{(l+1)}$
$y_3^{(l)}$
$\mathbf{w}_i^{(l+1)}$
$z_i^{(l+1)}$
$f$
$y_i^{(l+1)}$
$y_2^{(l)}$
$y_1^{(l)}$


+1
$b_i^{(l+1)}$
$r_3^{(l)}$
$y_3^{(l)}$
$\tilde{y}_3^{(l)}$
$r_2^{(l)}$
$\mathbf{w}_i^{(l+1)}$
$z_i^{(l+1)}$
$f$
$y_i^{(l+1)}$
$y_2^{(l)}$
$\tilde{y}_2^{(l)}$
$r_1^{(l)}$
$y_1^{(l)}$
$\tilde{y}_1^{(l)}$

# Dropout



(a) Standard Neural Net



(b) After applying dropout.

# Logits layer

- The final layer in our neural network is the logits layer
  - Returns the raw values for our predictions.
    - ✓ logits = tf.layers.dense(inputs=dropout, units=10)

# Loss calculation

```
loss = None
train_op = None

# Calculate loss for both TRAIN and EVAL modes
if mode != learn.ModeKeys.INFER:
  onehot_labels = tf.one_hot(indices=tf.cast(labels, tf.int32), depth=10)
  loss = tf.losses.softmax_cross_entropy(
      onehot_labels=onehot_labels, logits=logits)
```

# Configuring training

```
# Configure the Training Op (for TRAIN mode)
if mode == learn.ModeKeys.TRAIN:
    train_op = tf.contrib.layers.optimize_loss(
        loss=loss,
        global_step=tf.contrib.framework.get_global_step(),
        learning_rate=0.001,
        optimizer="SGD")
```

# Predictions

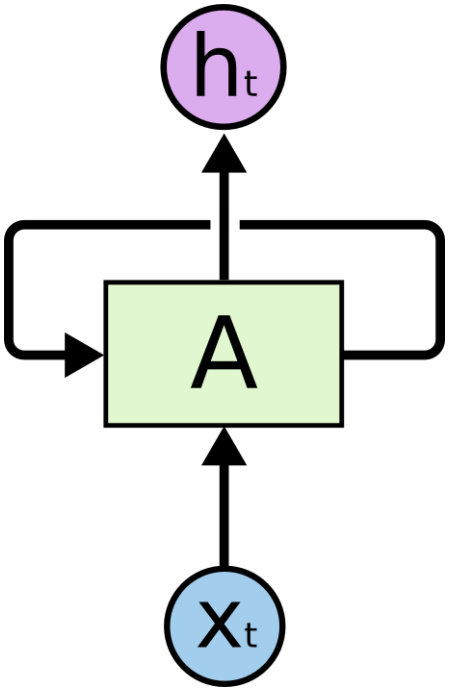
```
predictions = {
    "classes": tf.argmax(
        input=logits, axis=1),
    "probabilities": tf.nn.softmax(
        logits, name="softmax_tensor")
}
# Return a ModelFnOps object
return model_fn_lib.ModelFnOps(
    mode=mode, predictions=predictions, loss=loss, train_op=train_op)
```

# Recurrent Neural Networks

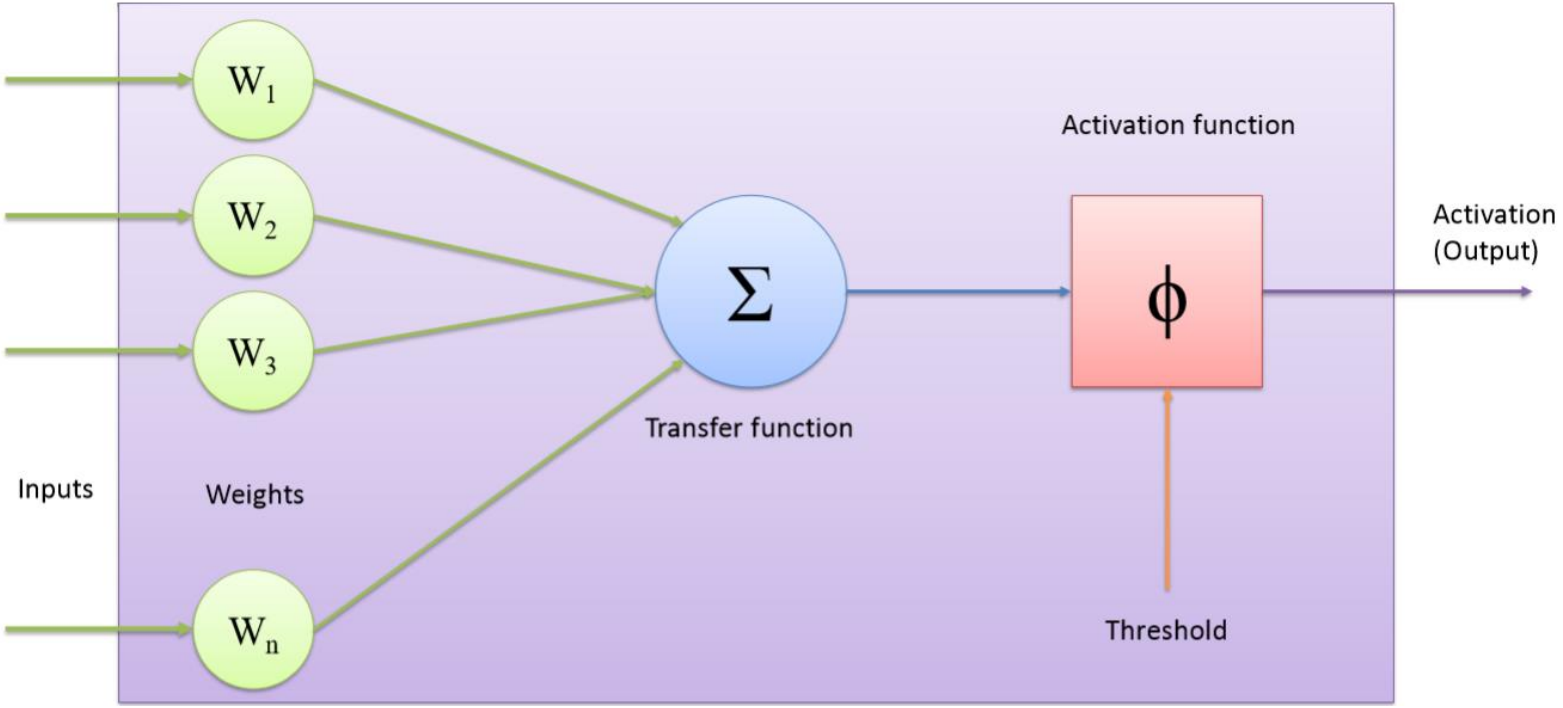
- What do we miss using CNN, FCN, DetectNet and other ANNs?

# Recurrent Neural Networks

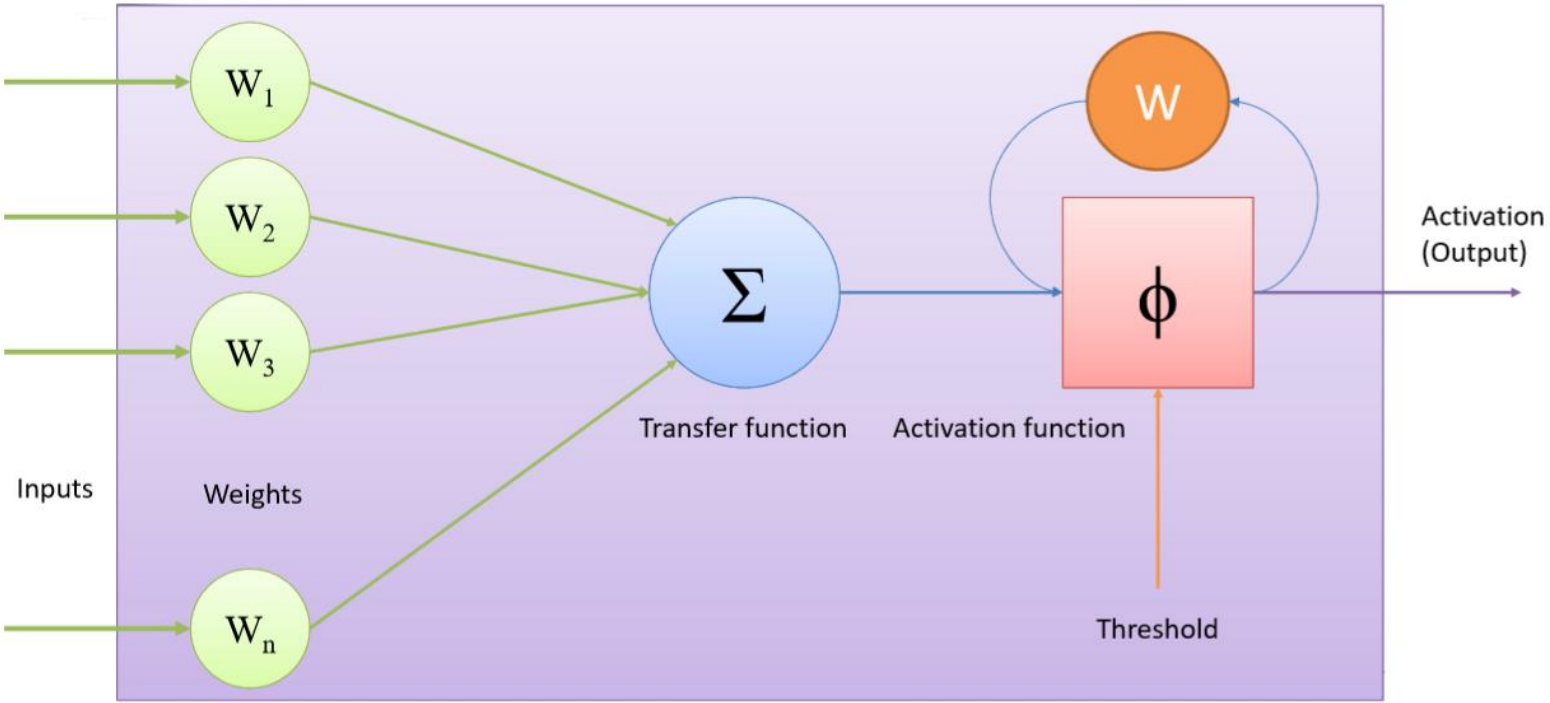
- Answer:
  - We miss time dimension
  - Unable to process sequences taking into account the previous results



# Artificial Neuron

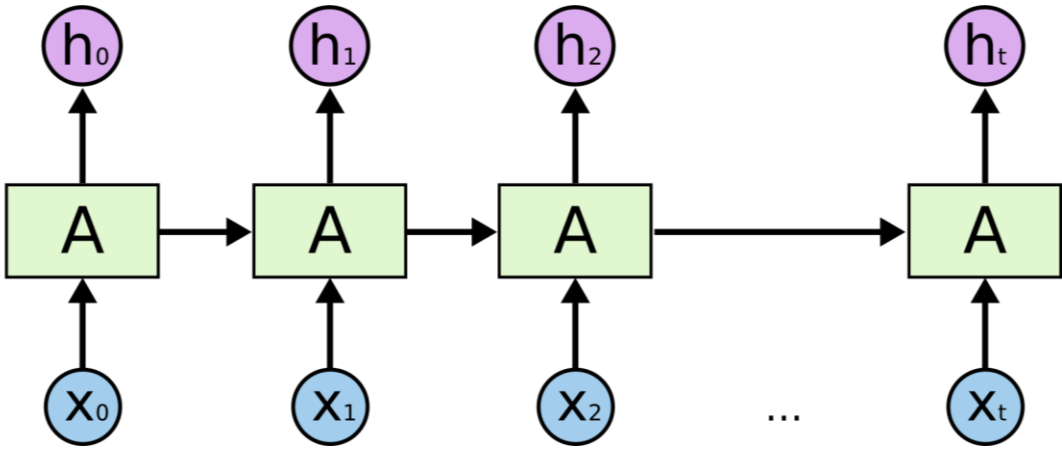


# Recurrent Artificial Neuron



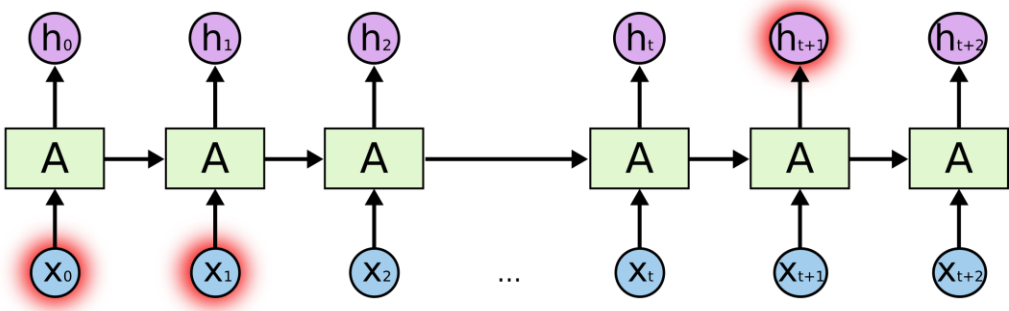
# RNN

- These loops make recurrent neural networks seem difficult to train.
- A recurrent neural network can be thought of as multiple copies of the same network, each passing a message to a successor.
- Consider what happens if we unroll the loop

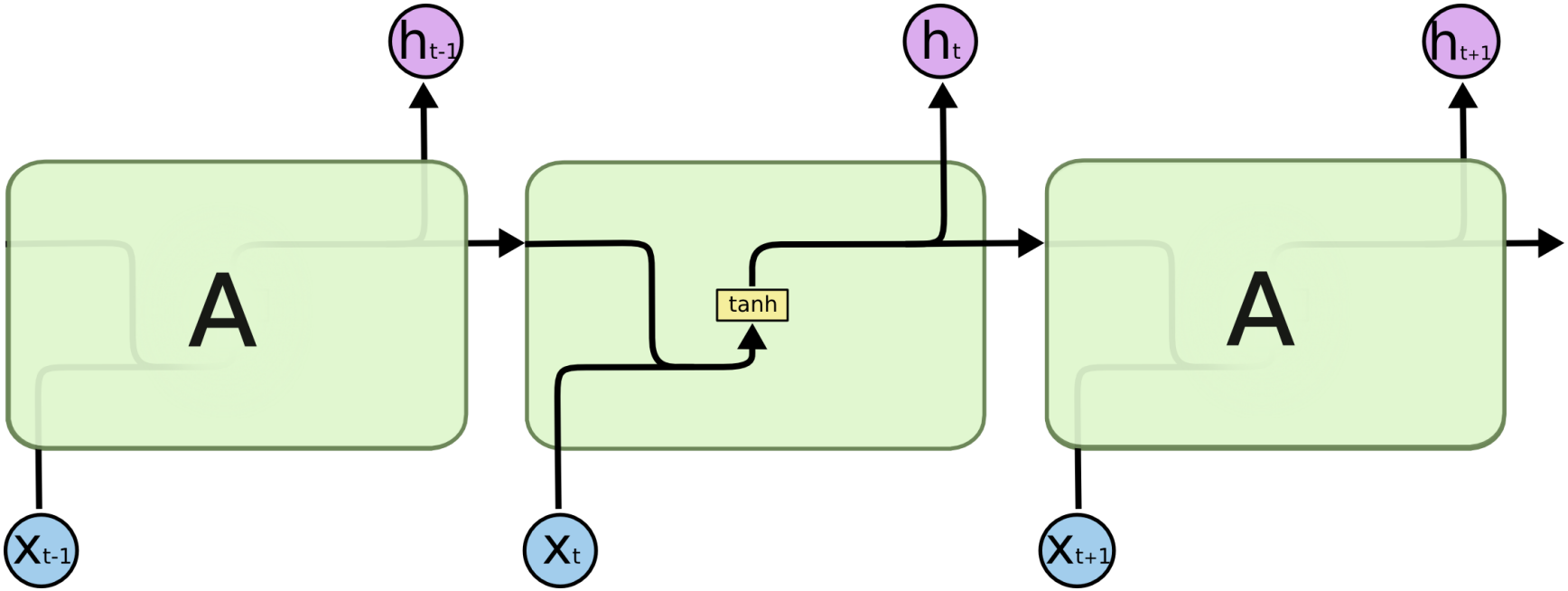


# RNN

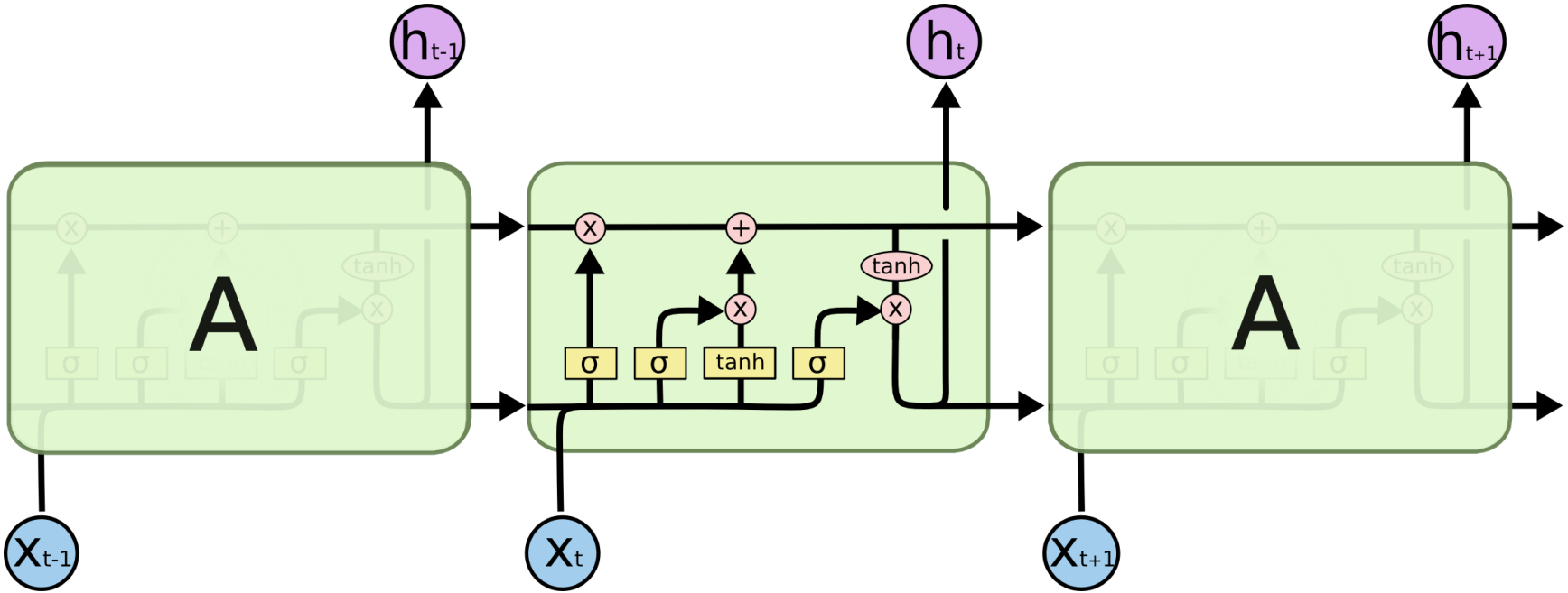
- RNN is able to connect previous information to the present task
  - Model trying to predict the next word based on the previous ones.
  - Depends on the gap between the currently predicted word and the context needed



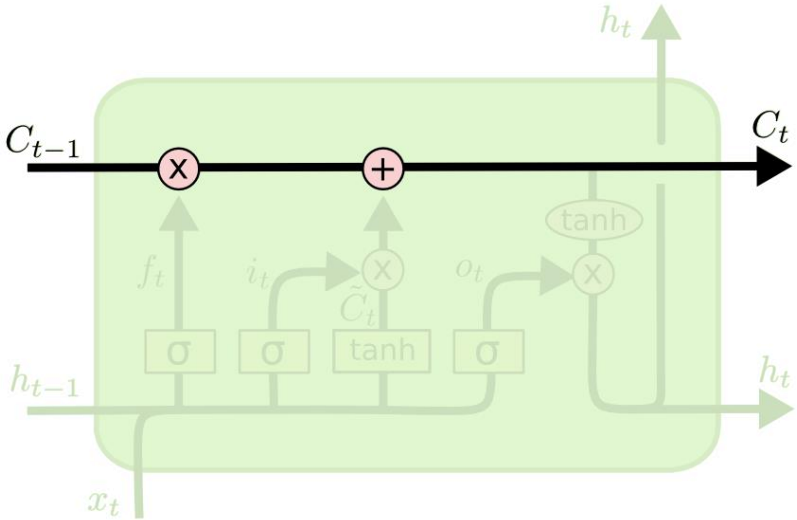
# RNN



# LSTM



# LSTM



# LSTM

```
lstm = tf.contrib.rnn.BasicLSTMCell(lstm_size)
# Initial state of the LSTM memory.
state = tf.zeros([batch_size, lstm.state_size])
probabilities = []
loss = 0.0
for current_batch_of_words in words_in_dataset:
    # The value of state is updated after processing each batch of words.
    output, state = lstm(current_batch_of_words, state)

    # The LSTM output can be used to make next word predictions
    logits = tf.matmul(output, softmax_w) + softmax_b
    probabilities.append(tf.nn.softmax(logits))
    loss += loss_function(probabilities, target_words)
```

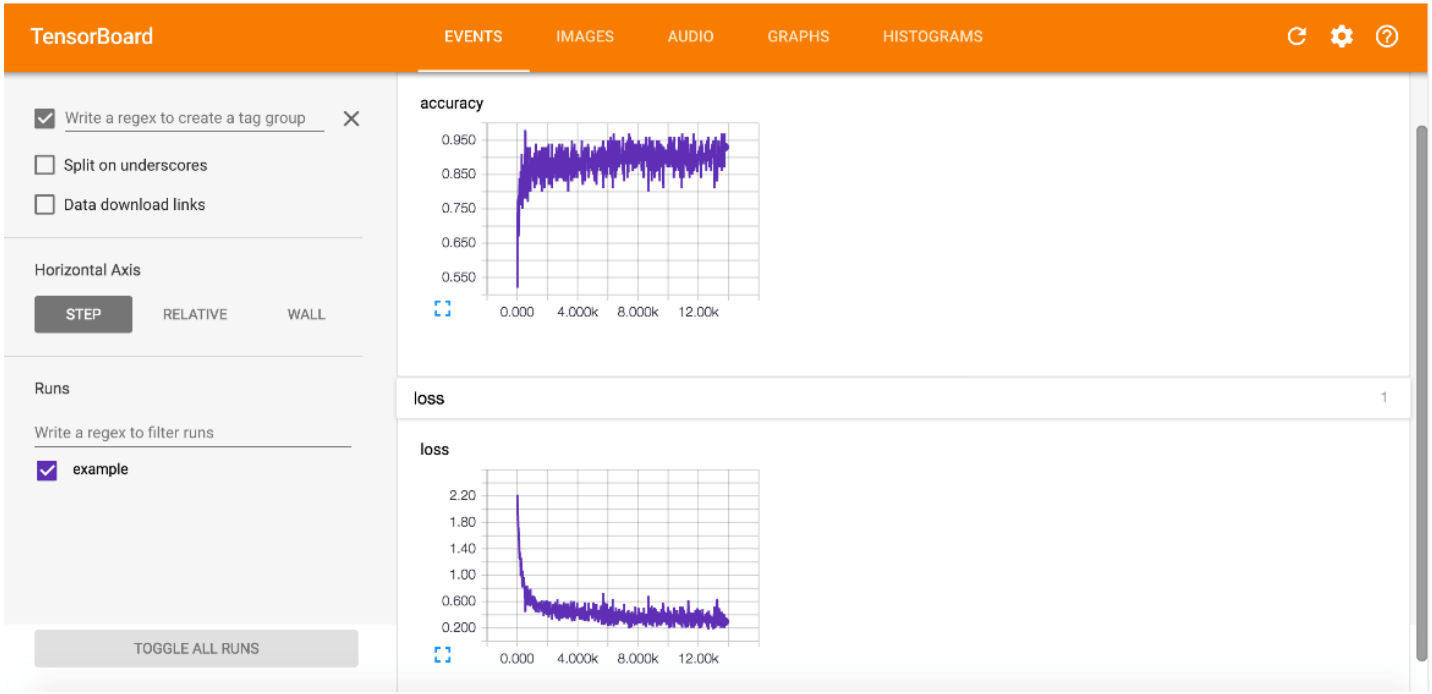
# LSTM

```
# Placeholder for the inputs in a given iteration.
words = tf.placeholder(tf.int32, [batch_size, num_steps])

lstm = tf.contrib.rnn.BasicLSTMCell(lstm_size)
# Initial state of the LSTM memory.
initial_state = state = tf.zeros([batch_size, lstm.state_size])

for i in range(num_steps):
    # The value of state is updated after processing each batch of words.
    output, state = lstm(words[:, i], state)

    # The rest of the code.
    # ...
```

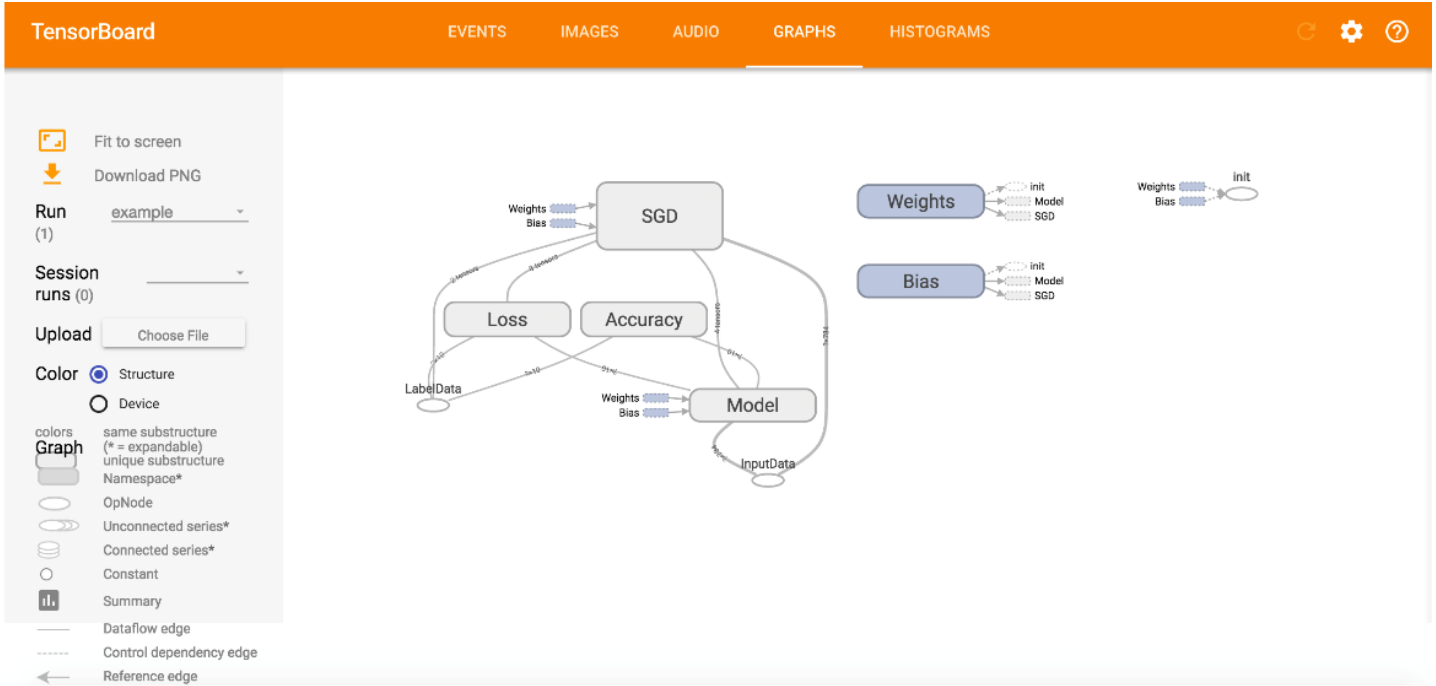
# LSTM

```
final_state = state
And this is how to implement an iteration over the whole dataset:

# A numpy array holding the state of LSTM after each batch of words.
numpy_state = initial_state.eval()
total_loss = 0.0
for current_batch_of_words in words_in_dataset:
    numpy_state, current_loss = session.run([final_state, loss],
        # Initialize the LSTM state from the previous iteration.
        feed_dict={initial_state: numpy_state, words:
current_batch_of_words})
    total_loss += current_loss
```

# TensorBoard



# TensorBoard



# TensorBoard

--> tensorboard --logdir=/tmp/tensorflow_logs

Then open http://0.0.0.0:6006/ into your web browser

# Device Placement

```
# Creates a graph.
a = tf.constant([1.0, 2.0, 3.0, 4.0, 5.0, 6.0], shape=[2, 3],
name='a')
b = tf.constant([1.0, 2.0, 3.0, 4.0, 5.0, 6.0], shape=[3, 2],
name='b')
c = tf.matmul(a, b)
# Creates a session with log_device_placement set to True.
sess =
tf.Session(config=tf.ConfigProto(log_device_placement=True))
# Runs the op.
print(sess.run(c))
```

# Manual Device Placement

```
# Creates a graph.
with tf.device('/cpu:0'):
  a = tf.constant([1.0, 2.0, 3.0, 4.0, 5.0, 6.0], shape=[2, 3],
name='a')
  b = tf.constant([1.0, 2.0, 3.0, 4.0, 5.0, 6.0], shape=[3, 2],
name='b')
  c = tf.matmul(a, b)
# Creates a session with log_device_placement set to True.
sess =
tf.Session(config=tf.ConfigProto(log_device_placement=True))
# Runs the op.
print(sess.run(c))
```