Deep Learning Lab

Paulo Rauber

paulo@idsia.ch

Imanol Schlag

imanol@idsia.ch

November 16, 2017

Overview

- Sessions: Mondays, 13:30 - 15:30, SI. 006:
  - September: ~~18~~, 25
  - October: 2, 9, 16, 23, 30
  - November: 6, 13, 20, 27
  - December: 4, 11, 18
  - January: ~~15 – 26~~
- Format: lectures and practical sessions
- Grading: two small projects (20%) and final project (80%)

Python

- High-level multi-paradigm programming language
- Additional reading:
  - Dive into Python 3 [Pilgrim, 2011]
  - PEP 8 – Style Guide for Python Code [Rossum et al., 2001]
  - A Guide to NumPy/SciPy documentation [Num, 2017a]

Virtualenv

- virtualenv: a tool to create isolated python environments

```
virtualenv --system-site-packages -p python3 test_env #creates environment
echo $PATH #original directories to search for executable files
source test_env/bin/activate #activates environment
echo $PATH #starts with /path/to/test_env/bin, containing python3 and pip3
pip3 install numpy #installs numpy for the current environment
python3 #this interpreter should be able to import numpy
deactivate
python3 #default interpreter (unaffected)
```

NumPy

- NumPy: scientific computing library for Python
  - powerful multidimensional arrays
  - efficient numerical computations
  - sophisticated broadcasting

- Additional reading:
  - NumPy Quickstart tutorial [Num, 2017c]
  - NumPy Broadcasting [Num, 2017b]

Slurm

- Slurm: workload manager for the ICS cluster [ICS, 2017]
  - Connect to hpc.ics.usi.ch using SSH
  - Use squeue to view jobs in the queue
  - Use scancel to send signals to jobs in the queue
  - Use sbatch to run a script that schedules a job. Example: [1]

```
#!/bin/bash -l
#
#SBATCH --job-name="abc"
#SBATCH --partition=tflow
#SBATCH --time=00:15:00
#SBATCH --output=abc.%j.out
#SBATCH --error=abc.%j.err

module load python/3.5.0
module load cudnn/8.0

srun python3 abc.py
```

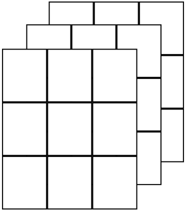
[1] Important: never run experiments directly on hpc.ics.usi.ch.

TensorFlow

- TensorFlow: library for numerical computations using data flow graphs
  - Scalable and multi-platform: from mobile devices to clusters
  - Enables transparent GPU usage
  - Widely used by researchers and practitioners
  - Python API

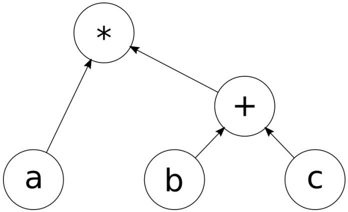
Tensor

- Tensor: for our purposes, synonymous with multidimensional array
- Examples of tensors:
  - Rank-0 tensor: real number
  - Rank-1 tensor: array of real numbers (real vector)
  - Rank-2 tensor: array of real vectors (real matrix)
  - Rank-3 tensor: array of real matrices



Computational graph

- Computational graph consists on nodes and edges:
  - Node: operation that receives zero or more tensors and produces zero or more tensors
  - Edge: connection between node outputs and node inputs



- Note: a constant is represented by a node that receives zero inputs and outputs the desired tensor.

Example: $\mathbf{a} \odot (\mathbf{b} + \mathbf{c})$

```python
import tensorflow as tf


def main():
    # Including constants in the default graph (nodes)
    a = tf.constant([2, 3, 5], dtype=tf.float32)
    b = tf.constant([1, 1, 3], dtype=tf.float32)
    c = tf.constant([1, 2, 2], dtype=tf.float32)

    # Including operations in the default graph (nodes)
    b_plus_c = tf.add(b, c)
    result = tf.multiply(a, b_plus_c)

    # Using operator overloading, we could accomplish the same by writing
    # result = a * (b + c)

    # Creating a TensorFlow session
    session = tf.Session()

    # Using the session to obtain the output for node `result`
    output = session.run(result)  # np.array([4., 9., 25.])

    print(output)

    session.close()

if __name__ == "__main__":
    main()
```

Session

- Session: responsible for managing resources to evaluate nodes

- Device management is possible (but often unnecessary):

```python
# ...
with tf.device('/gpu:0'):
    # Including constants in the default graph (nodes)
    a = tf.constant([2, 3, 5], dtype=tf.float32)
    b = tf.constant([1, 1, 3], dtype=tf.float32)
    c = tf.constant([1, 2, 2], dtype=tf.float32)

    # Including operations in the default graph (nodes)
    b_plus_c = tf.add(b, c)
    result = tf.multiply(a, b_plus_c)
# ...
```

- Additional reading:
    - TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems [Abadi et al., 2015]
    - TensorFlow: Using GPUs [Tf1, 2017c]

Variables

- Variable instance:
  - Represents state by a tensor
  - Usable as operand
- Variable instance adds several nodes to the graph:
  - Node that outputs initial state
  - Node that changes variable state as a side effect (assign node)
  - Node that outputs the current state (variable node)

Example: variables

```python
def main():
    a = tf.Variable([1.0, 1.0, 1.0], dtype=tf.float32)  # Variable
    b = tf.constant([1.0, 2.0, 3.0], dtype=tf.float32)

    c = a * b

    # Operation that assigns initial values to all variables (in our case, `a`)
    initialize = tf.global_variables_initializer()

    # Operation that assigns 2*a to `a`
    assign_double = tf.assign(a, 2 * a)

    session = tf.Session()

    # Obtains `initialize` output. Side effect: initializing `a`
    session.run(initialize)
    print(session.run(c))  # np.array([1.0, 2.0, 3.0])

    # Obtains `assign_double` output. Side effect: doubling `a`
    session.run(assign_double)
    print(session.run(c))  # np.array([2.0, 4.0, 6.0])
    session.run(assign_double)
    print(session.run(c))  # np.array([4.0, 8.0, 12.0])

    session.close()

    session = tf.Session()
    session.run(initialize)
    print(session.run(c))  # np.array([1.0, 2.0, 3.0])
    session.close()
```

Placeholders

- Placeholder: unknown tensor during graph creation
  - Usable as operand
  - Must be provided through the feed mechanism

```python
def main():
    a = tf.constant([1.0, 2.0, 3.0], dtype=tf.float32)
    b = tf.placeholder(dtype=tf.float32)  # Placeholder, shape omitted

    c = a * b

    session = tf.Session()

    print(session.run(c, feed_dict={b: 2.0}))  # np.array([2.0, 4.0, 6.0])
    print(session.run(c, feed_dict={b: [1.0, 2.0, 3.0]}))  # np.array([1.0, 4.0, 9.0])

    session.close()
```

Gradients

- tf.gradient: outputs partial derivative of a scalar with respect to each element of a tensor [2].
- Example:

$$y = \sum_{i=1}^{3} x_i^2 \implies \frac{\partial y}{\partial x_j} = 2x_j.$$

```python
def main():
    x = tf.Variable([1.0, 2.0, 3.0])
    y = tf.reduce_sum(tf.square(x))

    grad = tf.gradients(y, x)[0]  # Gradient of y wrt `x`

    initializer = tf.global_variables_initializer()

    session = tf.Session()
    session.run(initializer)
    print(session.run(grad))  # np.array([2.0, 4.0, 6.0])
    session.close()
```

[2] The function is much more general. See the documentation for details.

Gradient descent

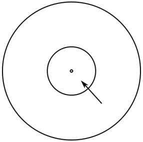
- Consider the task of minimizing $f : \mathbb{R}^D \to \mathbb{R}$.

- Gradient descent starts at an arbitrary estimate $\mathbf{x}_0 \in \mathbb{R}^D$ and iteratively updates this estimate using

$$\mathbf{x}_{t+1} = \mathbf{x}_t - \eta_t \nabla f(\mathbf{x}_t),$$

  where η_t is the learning rate at iteration t.

- Gradient descent converges to a critical point for appropriate η_t.

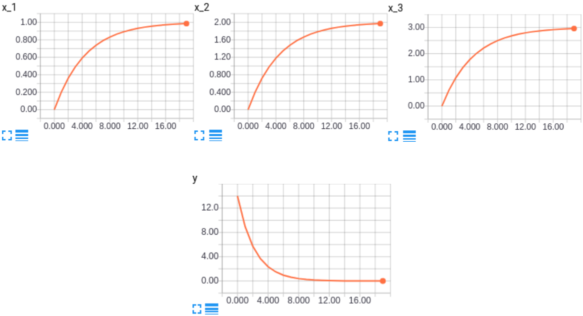


Example: minimizing $\sum_{i=1}^{3}(x_i - i)^2$ wrt $\mathbf{x}$

```python
def main():
    n_iterations = 20

    learning_rate = tf.constant(1e-1, dtype=tf.float32)

    # Goal: finding x such that y is minimum
    x = tf.Variable([0.0, 0.0, 0.0])  # Initial guess
    y = tf.reduce_sum(tf.square(x - tf.constant([1.0, 2.0, 3.0])))

    grad = tf.gradients(y, x)[0]

    update = tf.assign(x, x - learning_rate * grad)  # Gradient descent update

    initializer = tf.global_variables_initializer()

    session = tf.Session()
    session.run(initializer)

    for _ in range(n_iterations):
        session.run(update)
        print(session.run(x))  # State of `x` at this iteration

    session.close()
```

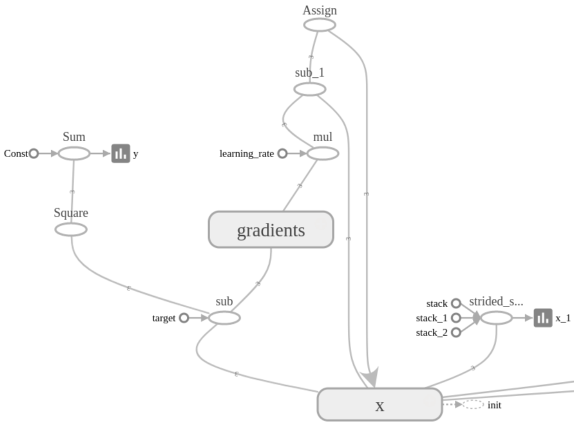
TensorBoard

- TensorBoard: visualizing summary data



TensorBoard

- TensorBoard: visualizing computational graph



TensorBoard

```
def main():
    directory = '/tmp/gradient_descent'  # Directory for data storage
    os.makedirs(directory)

    n_iterations = 20

    # Naming constants/variables to facilitate inspection
    learning_rate = tf.constant(1e-1, dtype=tf.float32, name='learning_rate')
    x = tf.Variable([0.0, 0.0, 0.0], name='x')
    target = tf.constant([1.0, 2.0, 3.0], name='target')
    y = tf.reduce_sum(tf.square(x - target))

    grad = tf.gradients(y, x)[0]

    update = tf.assign(x, x - learning_rate * grad)

    tf.summary.scalar('y', y)  # Includes summary attached to `y`
    tf.summary.scalar('x_1', x[0])  # Includes summary attached to `x[0]`
    tf.summary.scalar('x_2', x[1])  # Includes summary attached to `x[1]`
    tf.summary.scalar('x_3', x[2])  # Includes summary attached to `x[2]`

    # Merges all summaries into single a operation
    summaries = tf.summary.merge_all()

    initializer = tf.global_variables_initializer()

    # next slide ...
```

TensorBoard

```python
# ... previous slide
session = tf.Session()

# Creating object that writes graph structure and summaries to disk
writer = tf.summary.FileWriter(directory, session.graph)

session.run(initializer)

for t in range(n_iterations):
    # Updates `x` and obtains the summaries for iteration t
    s, _ = session.run([summaries, update])

    # Stores the summaries for iteration t
    writer.add_summary(s, t)

print(session.run(x))

writer.close()
session.close()

# Run tensorboard --logdir="/tmp/gradient_descent" --port 6006
# Access http://localhost:6006 and see scalars/graphs
```

TensorFlow

- Additional reading
  - TensorFlow: Develop [Tf1, 2017a]
    - Get Started
    - Programmer's Guide
    - Tutorials
    - Performance
  - TensorFlow for deep learning research [Tf1, 2017b]
  - TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems [Abadi et al., 2015]

Linear regression: model

- Consider an iid dataset $\mathcal{D} = (\mathbf{x}_1, y_1), \dots, (\mathbf{x}_N, y_N)$, where $\mathbf{x}_i \in \mathbb{R}^D$ and $y_i \in \mathbb{R}$.
- Regression: predicting target y given new observation $\mathbf{x}$
- Simple model:

$$y = \mathbf{w}\mathbf{x} = \sum_{j=1}^{D} w_j x_j$$

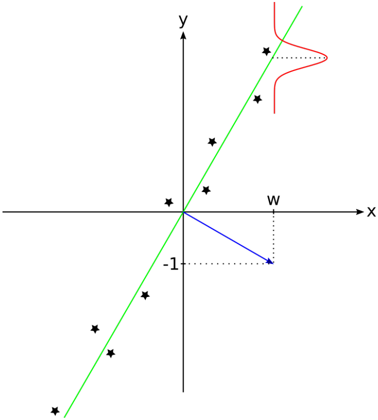
- Linear regression (without a bias term):

$$p(y \mid \mathbf{x}, \mathbf{w}) = \mathcal{N}(y \mid \mathbf{w}\mathbf{x}, \sigma^2),$$
$$\mathbb{E}[Y \mid \mathbf{x}, \mathbf{w}] = \mathbf{w}\mathbf{x}$$

Linear regression: geometry for $D = 1$

- The solutions to $wx - y = 0$ constitute a hyperplane

$$\{(x, y) \mid (w, -1) \cdot (x, y) = 0\}$$



Linear regression: likelihood

- Assuming constant σ^2, the conditional likelihood is given by

$$p(\mathcal{D} \mid \mathbf{w}) = \prod_{i=1}^{N} \mathcal{N}(y_i \mid \mathbf{w}\mathbf{x}_i, \sigma^2)$$

- The log-likelihood is given by

$$\log p(\mathcal{D} \mid \mathbf{w}) = -\frac{N}{2} \log 2\pi\sigma^2 - \frac{1}{2\sigma^2} \sum_{i=1}^{N} (y_i - \mathbf{w}\mathbf{x}_i)^2$$

- Maximizing the likelihood wrt $\mathbf{w}$ corresponds to minimizing

$$J = \frac{1}{N} \sum_{i=1}^{N} (y_i - \mathbf{w}\mathbf{x}_i)^2$$

Linear regression: extensions

- If $\mathbf{w}$ maximizes the likelihood, we may predict $y = \mathbf{w}\mathbf{x}$ given $\mathbf{x}$
  - Alternative: maximum a posteriori estimate (requires a prior)
  - Bayesian alternative: using a posterior predictive distribution
- Using a feature map $\boldsymbol{\phi} : \mathbb{R}^D \to \mathbb{R}^{D'}$:

$$p(y \mid \mathbf{x}, \mathbf{w}) = \mathcal{N}(y \mid \mathbf{w}\phi(\mathbf{x}), \sigma^2)$$

  - Bias-including feature map: $\phi(\mathbf{x}) = (\mathbf{x}, 1)$
    - $\mathbf{w}\phi(\mathbf{x}) = \mathbf{w}_{1:D}\mathbf{x} + w_{D+1}$
  - Polynomial feature map ($D = 1$): $\phi(x) = (1, x^1, \dots, x^{D'-1})$
    - $\mathbf{w}\phi(x) = \sum_{j=1}^{D'} w_j x^{j-1}$

Linear regression: additional reading

- Pattern Recognition and Machine Learning (Chapter 3) [Bishop, 2006]
- Machine Learning: a Probabilistic Perspective (Chapter 7) [Murphy, 2012]
- Notes on Machine Learning (Section 7) [Rauber, 2016]

Linear regression: example

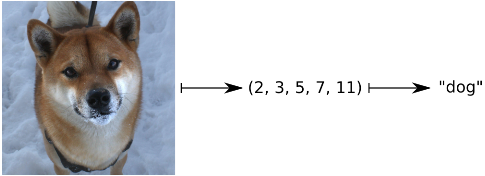
```python
def create_dataset(sample_size, n_dimensions, sigma, seed=None):
    """Create linear regression dataset (without bias term)"""
    random_state = np.random.RandomState(seed)

    # True weight vector: np.array([1, 2, ..., n_dimensions])
    w = np.arange(1, n_dimensions + 1)
    # Randomly generating observations
    X = random_state.uniform(-1, 1, (sample_size, n_dimensions))
    # Computing noisy targets
    y = np.dot(X, w) + random_state.normal(0.0, sigma, sample_size)

    return X, y


def main():
    sample_size_train = 100
    sample_size_val = 100

    n_dimensions = 10
    sigma = 0.1

    n_iterations = 20
    learning_rate = 0.5

    # Placeholder for the data matrix, where each observation is a row
    X = tf.placeholder(tf.float32, shape=(None, n_dimensions))
    # Placeholder for the targets
    y = tf.placeholder(tf.float32, shape=(None,))

    # next slide ...
```

Linear regression: example

```python
# ... previous slide
# Variable for the model parameters
w = tf.Variable(tf.zeros((n_dimensions, 1)), trainable=True)

# Loss function
prediction = tf.reshape(tf.matmul(X, w), (-1,))
loss = tf.reduce_mean(tf.square(y - prediction))

optimizer = tf.train.GradientDescentOptimizer(learning_rate)
train = optimizer.minimize(loss)  # Gradient descent update operation

initializer = tf.global_variables_initializer()

X_train, y_train = create_dataset(sample_size_train, n_dimensions, sigma)

session = tf.Session()
session.run(initializer)

for t in range(1, n_iterations + 1):
    l, _ = session.run([loss, train], feed_dict={X: X_train, y: y_train})
    print('Iteration {0}. Loss: {1}.'.format(t, l))

X_val, y_val = create_dataset(sample_size_val, n_dimensions, sigma)
l = session.run(loss, feed_dict={X: X_val, y: y_val})
print('Validation loss: {0}.'.format(l))

print(session.run(w).reshape(-1))

session.close()
```

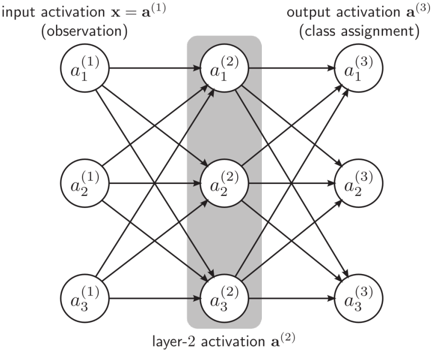
Classification task

- Consider an iid dataset $\mathcal{D} = (\mathbf{x}_1, \mathbf{y}_1), \dots, (\mathbf{x}_N, \mathbf{y}_N)$, where $\mathbf{x}_i \in \mathbb{R}^D$, and $\mathbf{y}_i \in \{0, 1\}^C$
- Given a pair $(\mathbf{x}, \mathbf{y}) \in \mathcal{D}$, we assume $y_j = 1$ if and only if observation $\mathbf{x}$ belongs to class j
- Each observation belongs to a single class
- Classification: predicting class assignment $\mathbf{y}$ given new observation $\mathbf{x}$



Feedforward neural network (MLP)

- Let L be the number of layers in the network



- Let $N^{(l)}$ be the number of neurons in layer l
- Input neurons, hidden neurons, output neurons

Feedforward neural network (MLP)

- Weighted input to neuron j in layer $l > 1$:

$$z_j^{(l)} = b_j^{(l)} + \sum_{k=1}^{N^{(l-1)}} w_{j,k}^{(l)} a_k^{(l-1)},$$

- Activation of neuron j in layer $1 < l < L$:

$$a_j^{(l)} = \sigma(z_j^{(l)}),$$

where σ is a differentiable function, such as $\sigma(z) = \frac{1}{1+e^{-z}}$

Feedforward neural network (MLP)

- Alternatively, the output of each layer $1 < l < L$ can be written as

$$\mathbf{a}^{(l)} = \sigma(\mathbf{W}^{(l)}\mathbf{a}^{(l-1)} + \mathbf{b}^{(l)}),$$

  where the activation function is applied element-wise
- The (softmax) activation of output neuron j is given by

$$a_j^{(L)} = \frac{e^{z_j^{(L)}}}{\sum_{k=1}^{C} e^{z_k^{(L)}}}.$$

- The output given $\mathbf{a}^{(1)} = \mathbf{x}$ is simply $\mathbf{a}^{(L)}$

Feedforward neural network (MLP)

- Let $\boldsymbol{\theta}$ represent an assignment to weights and biases
- Maximizing the likelihood $p(\mathcal{D} \mid \boldsymbol{\theta})$ corresponds to minimizing

$$J = -\frac{1}{N} \log p(\mathcal{D} \mid \boldsymbol{\theta}) = -\frac{1}{N} \sum_{(\mathbf{x},\mathbf{y}) \in \mathcal{D}} \sum_{k=1}^{C} y_k \log a_k^{(L)}$$

  with respect to $\boldsymbol{\theta}$
- The gradient $\nabla J(\boldsymbol{\theta})$ can be computed using backpropagation
- Minimization can be attempted by (stochastic) gradient descent or related techniques [Ruder, 2016]

Feedforward neural network (MLP)

- Additional reading:
  - Pattern Recognition and Machine Learning (Chapter 5) [Bishop, 2006]
  - Machine Learning: a Probabilistic Perspective (Section 16.5) [Murphy, 2012]
  - Neural networks and deep learning (Chapter 1) [Nielsen, 2015]
  - Notes on neural networks (Section 2) [Rauber, 2015]
  - Notes on machine learning (Section 17) [Rauber, 2016]

Example: MNIST classification

```python
def main():
    tf.set_random_seed(seed=None)

    # Downloads and loads MNIST dataset
    mnist = input_data.read_data_sets('/tmp/mnist/', one_hot=True)
    val_size = mnist.validation.num_examples

    # Training procedure hyperparameters
    learning_rate = 1e-2
    batch_size = 64
    n_epochs = 16
    verbose_freq = 2000

    # Model hyperparameters
    n_neurons_1 = 784  # Number of input neurons (28 x 28 x 1)
    n_neurons_2 = 100  # Number of neurons in the second layer (first hidden)
    n_neurons_3 = 100  # Number of neurons in the third layer (second hidden)
    n_neurons_4 = 10  # Number of output neurons (and classes)

    X = tf.placeholder(tf.float32, [None, n_neurons_1])
    Y = tf.placeholder(tf.float32, [None, n_neurons_4])

    # Model parameters. Important: should not be initialized to zero
    W2 = tf.Variable(tf.truncated_normal([n_neurons_1, n_neurons_2]))
    W3 = tf.Variable(tf.truncated_normal([n_neurons_2, n_neurons_3]))
    W4 = tf.Variable(tf.truncated_normal([n_neurons_3, n_neurons_4]))

    b2 = tf.Variable(tf.zeros(n_neurons_2))
    b3 = tf.Variable(tf.zeros(n_neurons_3))
    b4 = tf.Variable(tf.zeros(n_neurons_4))
```

Example: MNIST classification

```
# Model definition
# The rectified linear activation function is given by a = max(z, 0)
A2 = tf.nn.relu(tf.matmul(X, W2) + b2)
A3 = tf.nn.relu(tf.matmul(A2, W3) + b3)
Z4 = tf.matmul(A3, W4) + b4

# Loss definition
# Important: this function expects weighted inputs, not activations
loss = tf.nn.softmax_cross_entropy_with_logits(labels=Y, logits=Z4)
loss = tf.reduce_mean(loss)

hits = tf.equal(tf.argmax(Z4, axis=1), tf.argmax(Y, axis=1))
accuracy = tf.reduce_mean(tf.cast(hits, tf.float32))

# Using Adam instead of gradient descent
optimizer = tf.train.AdamOptimizer(learning_rate)
train = optimizer.minimize(loss)

# Allows saving model to disc
saver = tf.train.Saver()
```

Example: MNIST classification

```python
session = tf.Session()
session.run(tf.global_variables_initializer())

# Using mini-batches instead of entire dataset
n_batches = n_epochs * (mnist.train.num_examples // batch_size)  # roughly
for t in range(n_batches):
    X_batch, Y_batch = mnist.train.next_batch(batch_size)
    session.run(train, {X: X_batch, Y: Y_batch})

    # Computes validation loss every `verbose_freq` batches
    if verbose_freq > 0 and t % verbose_freq == 0:
        X_batch, Y_batch = mnist.validation.next_batch(val_size)
        l = session.run(loss, {X: X_batch, Y: Y_batch})

        print('Batch: {0}. Validation loss: {1}.'.format(t, l))

saver.save(session, '/tmp/mnist.ckpt')
session.close()

# Loading model from file
session = tf.Session()
saver.restore(session, '/tmp/mnist.ckpt')

# In a proper experiment, test set results are computed only once, and
# absolutely never considered during the choice of hyperparameters
X_batch, Y_batch = mnist.test.next_batch(val_size)
acc = session.run(accuracy, {X: X_batch, Y: Y_batch})
print('Test accuracy: {0}.'.format(acc))

session.close()
```

Convolutional neural network: overview

- Convolutional neural network (CNN):
  - Parameterized function
  - Parameters may be adapted to minimize a cost function using gradient descent
  - Suitable for image classification: explores the spatial relationships between pixels
  - Three important types of layers: convolutional layers, max-pooling layers, and fully connected layers

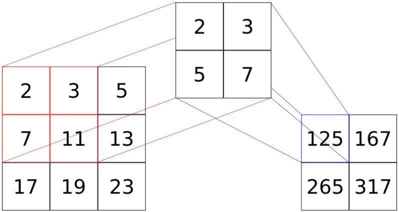
Convolutional neural network: notation

- Image: a function $\mathbf{f} : \mathbb{Z}^2 \to \mathbb{R}^c$
  - $\mathbf{a} \in \mathbb{Z}^2$ is a pixel
  - $\mathbf{f}(\mathbf{a})$ is the value of pixel $\mathbf{a}$
  - If $\mathbf{f}(\mathbf{a}) = (f_1(\mathbf{a}), \dots, f_c(\mathbf{a}))$, then f_i is channel i
  - Window $W \subset \mathbb{Z}^2$ is a finite set $W = [s_1, S_1] \times [s_2, S_2]$ that corresponds to a rectangle in the image domain
  - If the domain Z of an image $\mathbf{f}$ is a window, it is possible to flatten $\mathbf{f}$ into a vector $\mathbf{x} \in \mathbb{R}^{c|Z|}$
- Consider an iid dataset $\mathcal{D} = (\mathbf{x}_1, \mathbf{y}_1), \dots, (\mathbf{x}_N, \mathbf{y}_N)$, such that $\mathbf{x}_i \in \mathbb{R}^D$ and $\mathbf{y}_i \in \{0, 1\}^C$. Each vector $\mathbf{x}_i$ corresponds to a distinct image $\mathbb{Z}^2 \mapsto \mathbb{R}^c$, and all images are defined on the same window Z, such that $D = c|Z|$ [3]

[3] Notice that we denote the number of colors by c and the number of classes by C.

Convolutional layer

- A neuron in a convolutional layer is not necessarily connected to the activations of all neurons in the previous layer, but only to the activations in a particular $w \times h$ window W
- A neuron in a convolutional layer is replicated through parameter sharing for all windows of size $w \times h$ in the domain Z whose centers are offset by pre-defined steps (strides)



Convolutional layer

- Receives an input image $\mathbf{f}$ and outputs an image $\mathbf{o}$
- Each artificial neuron h in a convolutional layer l receives as input the values in a window $W = [s_1, S_1] \times [s_2, S_2] \subset Z$ of size $w \times h$, where Z is the domain of $\mathbf{f}$. The weighted input $z_h^{(l)}$ of that neuron is given by

$$z_h^{(l)} = b_h^{(l)} + \sum_{i=1}^{c} \sum_{j=s_1}^{S_1} \sum_{k=s_2}^{S_2} w_{h,i,j,k}^{(l)} a_{i,j,k}^{(l-1)},$$

  where $a_{i,j,k}^{(l-1)} = f_i(j, k)$ is the value of pixel (j, k) in channel i of the input image $\mathbf{f}$
- Activation function is typically rectified linear: $a_h^{(l)} = \max(0, z_h^{(l)})$

- An output image $\mathbf{o} : \mathbb{Z}^2 \to \mathbb{R}^n$ is obtained by replicating n neurons over the whole domain of the input image
- The activations corresponding to a neuron replicated in this way correspond to the values in a single channel of the output image $\mathbf{o}$ (appropriately arranged in $\mathbb{Z}^2$)
- The total number of free parameters in a convolutional layer is only $n(cwh + 1)$.

Convolutional layer

- If the parameters in a convolutional layer were not shared by replicated neurons, the number of parameters would be $mn(cwh + 1)$, where m is the number of windows of size $w \times h$ that fit into $\mathbf{f}$ (for the given strides)
- A convolutional layer is fully specified by the size of the filters (window size), the number of filters (number of channels in the output image), horizontal and vertical strides (which are usually 1)

Max-pooling layer

- Goal: achieving similar results to using comparatively larger convolutional filters in the next layers with less parameters
- Receives an input image $\mathbf{f} : \mathbb{Z}^2 \to \mathbb{R}^c$ and outputs an image $\mathbf{o} : \mathbb{Z}^2 \to \mathbb{R}^c$
- Reduces the size of the window domain Z of $\mathbf{f}$ by an operation that acts independently on each channel

$$o_i(j,k) = \max_{\mathbf{a} \in W_{j,k}} f_i(\mathbf{a}),$$

  where $i \in \{1, \dots, c\}$, $(j,k) \in \mathbb{Z}^2$, Z is the window domain of $\mathbf{f}$, and $W_{j,k} \subseteq Z$ is the input window corresponding to output pixel (j,k).
- A max-pooling layer is fully specified by the size of a pooling window and vertical/horizontal strides

Fully connected layer

- Receives a vector (or flattened image) and outputs a vector
- Analogous to a layer in a multilayer perceptron
- Typically only followed by other fully connected layers
- In a classification task, the output layer is typically fully connected with C neurons

Convolutional neural network

- Additional reading:
  - Pattern Recognition and Machine Learning (Chapter 5) [Bishop, 2006]
  - Machine Learning: a Probabilistic Perspective (Section 16.5) [Murphy, 2012]
  - Neural networks and deep learning (Chapter 6) [Nielsen, 2015]
  - Convolutional Neural Networks for Visual Recognition [Li and Karpathy, 2015]
  - Notes on neural networks (Section 5) [Rauber, 2015]
  - Notes on machine learning (Section 17) [Rauber, 2016]

Assignment 1: CNN MNIST classification

Adapt the MNIST digits classification example presented in the previous slides to use convolutional neural networks. You can use the architecture described in “Deep MNIST for Experts” as a reference [Tf1, 2017a].

Recurrent neural network: overview

- Recurrent neural network (RNN):
  - Parameterized function
  - Parameters may be adapted to minimize a cost function using gradient descent
  - Suitable for receiving a sequence of vectors and producing a sequence of vectors

Recurrent neural network: notation

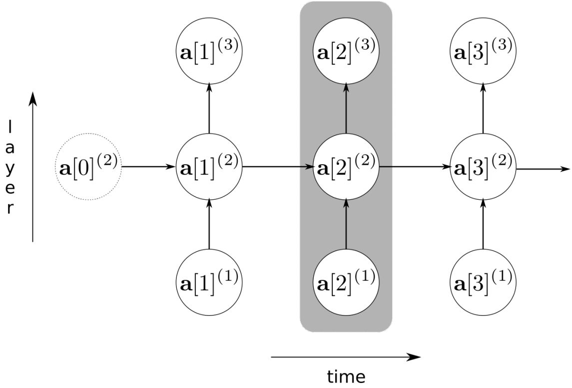
- Let A^+ denote the set of non-empty sequences of elements from the set A, and let $|X|$ denote the length of a sequence $X \in A^+$
- Let $X[t]$ denote the t-th element of sequence X
- Consider the dataset:

$$\mathcal{D} = \{(X_i, Y_i) \mid i \in \{1, \dots, N\}, X_i \in (\mathbb{R}^D)^+, Y_i \in (\mathbb{R}^C)^+\},$$

  and let $|X| = |Y|$ for every $(X, Y) \in \mathcal{D}$
- In words, the dataset $\mathcal{D}$ is composed of pairs (X, Y) of sequences of the same length. Each element of the two sequences is a real vector, but $X[t]$ and $Y[t]$ do not necessarily have the same dimension
- Sequence element classification: finding a function f that is able to generalize from $\mathcal{D}$

Recurrent neural network

- A recurrent neural network summarizes a sequence of vectors $X[1:t-1]$ into an activation vector
- This summary is combined with the input $X[t]$ to produce the output and the summary for the next timestep



Recurrent neural network

- We consider recurrent neural networks with a single recurrent layer and a softmax output layer
- In that case, the weighted input to neuron j in the recurrent layer at time step t is given by

$$z[t]_j^{(2)} = b_j^{(2)} + \sum_{k=1}^{N^{(1)}} w_{j,k}^{(2)} a[t]_k^{(1)} + \sum_{k=1}^{N^{(2)}} \omega_{j,k}^{(2)} a[t-1]_k^{(2)},$$

  where $\mathbf{a}[0]^{(l)}$ is usually zero (or learnable)
- The corresponding activation is given by $a[t]_j^{(2)} = \sigma(z[t]_j^{(2)})$
- The output activation $\mathbf{a}[t]^{(3)}$ is computed from $\mathbf{a}[t]^{(2)}$ as usual

Recurrent neural network

- The output of the recurrent neural network on input $X = \mathbf{a}[1]^{(1)}, \dots, \mathbf{a}[T]^{(1)}$ is the sequence $\mathbf{a}[1]^{(3)}, \dots, \mathbf{a}[T]^{(3)}$
- Intuitively, the sequence X is presented to the network element by element
- The network behaves similarly to a single hidden layer feedforward neural network, except for the fact that the output activation $\mathbf{a}[t]^{(2)}$ of the hidden layer at time t possibly affects the weighted input $\mathbf{z}[t+1]^{(2)}$ of the hidden layer at time $t+1$
- An ideal recurrent neural network would be capable of representing a sequence $X[1:t]$ by its hidden layer activation $\mathbf{a}[t]^{(2)}$ to allow correct classification of $X[t+1]$
- Parameters are shared across time

Recurrent neural network

- Consider a sequence element classification cost function J given by

$$J = -\frac{1}{NT} \sum_{(X,Y) \in \mathcal{D}} \sum_{t=1}^{T} \sum_{j=1}^{C} Y[t]_j \log a[t]_j^{(3)}$$

- Let $\boldsymbol{\theta}$ represent an assignment to weights and biases
- The gradient $\nabla J(\boldsymbol{\theta})$ can be computed using backpropagation through time
- Minimization can be attempted by (stochastic) gradient descent or related techniques [Ruder, 2016]

Recurrent neural network

- Additional reading:
  - Supervised sequence labelling with recurrent neural networks (Sec. 3.2) [Graves, 2012]
  - Notes on Neural networks (Sec. 6) [Rauber, 2015]
  - The Unreasonable Effectiveness of Recurrent Neural Networks [Karpathy, 2015]
  - Understanding LSTM Networks [Olah, 2015]
  - Recurrent Neural Networks in Tensorflow I [R2R, 2016]

Example: N-back using RNNs

```python
import numpy as np
import tensorflow as tf


def nback(n, k, length, random_state):
    """Creates n-back task given n, number of digits k, and sequence length.

    Given a sequence of integers `xi`, the sequence `yi` has yi[t] = 1 if and
    only if xi[t] == xi[t - n].
    """
    xi = random_state.randint(k, size=length)  # Input sequence
    yi = np.zeros(length, dtype=int)  # Target sequence

    for t in range(n, length):
        yi[t] = (xi[t - n] == xi[t])

    return xi, yi
```

Example: N-back using RNNs

```
def nback_dataset(n_sequences, mean_length, std_length, n, k, random_state):
    """Creates dataset composed of n-back tasks."""
    X, Y, lengths = [], [], []

    for _ in range(n_sequences):
        # Choosing length for current task
        length = random_state.normal(loc=mean_length, scale=std_length)
        length = int(max(n + 1, length))

        # Creating task
        xi, yi = nback(n, k, length, random_state)

        # Storing task
        X.append(xi)
        Y.append(yi)
        lengths.append(length)

    # Creating padded arrays for the tasks
    max_len = max(lengths)
    Xarr = np.zeros((n_sequences, max_len), dtype=np.int64)
    Yarr = np.zeros((n_sequences, max_len), dtype=np.int64)

    for i in range(n_sequences):
        Xarr[i, 0: lengths[i]] = X[i]
        Yarr[i, 0: lengths[i]] = Y[i]

    return Xarr, Yarr, lengths
```

```python
def main():
    seed = 0

    # Task parameters
    n = 3  # n-back
    k = 4  # Input dimension
    mean_length = 20 # Mean sequence length
    std_length = 5  # Sequence length standard deviation
    n_sequences = 512  # Number of training/validation sequences

    # Creating datasets
    random_state = np.random.RandomState(seed=seed)
    X_train, Y_train, lengths_train = nback_dataset(n_sequences, mean_length,
                                                    std_length, n, k,
                                                    random_state)

    X_val, Y_val, lengths_val = nback_dataset(n_sequences, mean_length,
                                              std_length, n, k, random_state)

    # Model parameters
    hidden_units = 64  # Number of recurrent units

    # Training procedure parameters
    learning_rate = 1e-2
    n_epochs = 256

    # Model definition
    X_int = tf.placeholder(shape=[None, None], dtype=tf.int64)
    Y_int = tf.placeholder(shape=[None, None], dtype=tf.int64)
    lengths = tf.placeholder(shape=[None], dtype=tf.int64)
```

Example: N-back using RNNs

```
batch_size = tf.shape(X_int)[0]
max_len = tf.shape(X_int)[1]

# One-hot encoding X_int
X = tf.one_hot(X_int, depth=k)  # shape: (batch_size, max_len, k)
# One-hot encoding Y_int
Y = tf.one_hot(Y_int, depth=2)  # shape: (batch_size, max_len, 2)

cell = tf.contrib.rnn.BasicRNNCell(num_units=hidden_units)
init_state = cell.zero_state(batch_size, dtype=tf.float32)

# rnn_outputs shape: (batch_size, max_len, hidden_units)
rnn_outputs, \
    final_state = tf.nn.dynamic_rnn(cell, X, sequence_length=lengths,
                                    initial_state=init_state)

# rnn_outputs_flat shape: ((batch_size * max_len), hidden_units)
rnn_outputs_flat = tf.reshape(rnn_outputs, [-1, hidden_units])

# Weights and biases for the output layer
Wout = tf.Variable(tf.truncated_normal(shape=(hidden_units, 2),
                                       stddev=0.1))
bout = tf.Variable(tf.zeros(shape=[2]))

# Z shape: ((batch_size * max_len), 2)
Z = tf.matmul(rnn_outputs_flat, Wout) + bout

Y_flat = tf.reshape(Y, [-1, 2])  # shape: ((batch_size * max_len), 2)
```

Example: N-back using RNNs

```
# Creates a mask to disregard padding
mask = tf.sequence_mask(lengths, dtype=tf.float32)
mask = tf.reshape(mask, [-1])  # shape: (batch_size * max_len)

# Network prediction
pred = tf.argmax(Z, axis=1) * tf.cast(mask, dtype=tf.int64)
pred = tf.reshape(pred, [-1, max_len])  # shape: (batch_size, max_len)

hits = tf.reduce_sum(tf.cast(tf.equal(pred, Y_int), tf.float32))
hits = hits - tf.reduce_sum(1 - mask)  # Disregards padding

# Accuracy: correct predictions divided by total predictions
accuracy = hits/tf.reduce_sum(mask)

# Loss definition (masking to disregard padding)
loss = tf.nn.softmax_cross_entropy_with_logits(labels=Y_flat, logits=Z)
loss = tf.reduce_sum(loss*mask)/tf.reduce_sum(mask)

optimizer = tf.train.AdamOptimizer(learning_rate)
train = optimizer.minimize(loss)
```

Example: N-back using RNNs

```
session = tf.Session()
session.run(tf.global_variables_initializer())

for e in range(1, n_epochs + 1):
    feed = {X_int: X_train, Y_int: Y_train, lengths: lengths_train}
    l, _ = session.run([loss, train], feed)
    print('Epoch: {0}. Loss: {1}.'.format(e, l))

feed = {X_int: X_val, Y_int: Y_val, lengths: lengths_val}
accuracy_ = session.run(accuracy, feed)
print('Validation accuracy: {0}.'.format(accuracy_))

# Shows first task and corresponding prediction
xi = X_val[0, 0: lengths_val[0]]
yi = Y_val[0, 0: lengths_val[0]]
print('Sequence:')
print(xi)
print('Ground truth:')
print(yi)
print('Prediction:')
print(session.run(pred, {X_int: [xi], lengths: [len(xi)]})[0])

session.close()
```

Long short-term memory network: overview

- Long short-term memory network (LSTM):
  - Parameterized function
  - Parameters may be adapted to minimize a cost function using gradient descent
  - Suitable for receiving a sequence of vectors and producing a sequence of vectors
  - Mitigates the <u>vanishing gradients problem</u>
  - Better than simple recurrent neural networks at learning dependencies between input and target vectors that manifest after many time steps

Long short-term memory network: overview

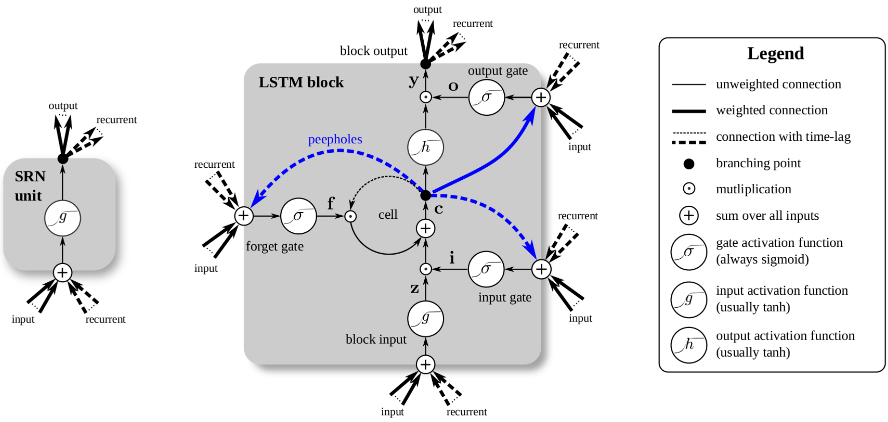


Image from [Greff et al., 2016]

Long short-term memory network

- We consider long short-term memory networks with a single hidden layer for the task of sequence element classification
- The input activation for the network at time t for $(X, Y) \in \mathcal{D}$ is defined as $X[t] = \mathbf{a}[t]^{(1)}$
- Similarly to a neuron in the hidden layer of a simple recurrent neural network, memory block j also receives the vectors $\mathbf{a}[t]^{(1)}$ and $\mathbf{a}[t-1]^{(2)}$ at time step t, and outputs a scalar $a[t]_j^{(2)}$
- However, the computations performed in a memory block are considerably more involved than those in a simple recurrent artificial neuron

Long short-term memory network

- A memory block is composed of four modules: cell, input gate I, forget gate F, and output gate O
- The weighted input $z[t]_j^{(2)}$ to the cell in memory block j is defined as

$$z[t]_j^{(2)} = b_j^{(2)} + \sum_{k=1}^{N^{(1)}} w_{j,k}^{(2)} a[t]_k^{(1)} + \sum_{k=1}^{N^{(2)}} \omega_{j,k}^{(2)} a[t-1]_k^{(2)},$$

  where $\mathbf{a}[0]^{(2)}$ may be zero
- This is analogous to the weighted input for neuron j in the hidden layer of a simple recurrent network

Long short-term memory network

- The activation $s[t]_j^{(2)}$ of the cell in memory block j is defined as

$$s[t]_j^{(2)} = a[t]_{F,j}^{(2)} s[t-1]_j^{(2)} + a[t]_{I,j}^{(2)} g(z[t]_j^{(2)}),$$

  where $\mathbf{s}[0]^{(2)}$ may be zero, and g is a differentiable activation function
- The terms $a[t]_{F,j}^{(2)}$ and $a[t]_{I,j}^{(2)}$ correspond to the activations of the forget and input gates, respectively, and will be defined shortly
- Because each of these two scalars is usually between 0 and 1, they control how much the previous activation of the cell and the current weighted input to the cell affect its current activation

Long short-term memory network

- The weighted input $z[t]^{(2)}_{G,j}$ of a gate $G = I, F$ or O in memory block j is defined as

$$z[t]^{(2)}_{G,j} = b^{(2)}_{G,j} + \psi^{(2)}_{G,j} s[t-1]^{(2)}_j + \sum_{k=1}^{N^{(1)}} w^{(2)}_{G,j,k} a[t]^{(1)}_k + \sum_{k=1}^{N^{(2)}} \omega^{(2)}_{G,j,k} a[t-1]^{(2)}_k,$$

  where $\psi_{G,j}$ is the so-called peephole weight
- The activation $a[t]^{(2)}_{G,j}$ of a gate G in memory block j is defined as $a[t]^{(2)}_{G,j} = f(z[t]^{(2)}_{G,j})$, where f is typically the sigmoid function
- Each gate G in memory block j has its own parameters and behaves similarly to a simple recurrent neuron

Long short-term memory network

- The output activation $a[t]_j^{(2)}$ of memory block j is defined as

$$a[t]_j^{(2)} = a[t]_{O,j}^{(2)} h(s[t]_j^{(2)}),$$

  where h is a differentiable activation function
- The activation of the output gate controls how much the current activation of the cell affects the output of the memory block
- A memory block can be interpreted as a parameterized circuit. By training the network, a memory block may learn when to store, output and erase its memory (cell activation), given the current input activation to the network and the previous activation of the memory blocks

Long short-term memory network

- The output activation $\mathbf{a}[t]^{(3)}$ is computed from $\mathbf{a}[t]^{(2)}$ as usual
- The output of the long short-term memory network on input $X = \mathbf{a}[1]^{(1)}, \dots, \mathbf{a}[T]^{(1)}$ is the sequence $\mathbf{a}[1]^{(3)}, \dots, \mathbf{a}[T]^{(3)}$
- An ideal LSTM would be capable of representing a sequence $X[1 : t]$ by the activation of its memory blocks $\mathbf{a}[t]^{(2)}$ and cells $\mathbf{s}[t]^{(2)}$ to allow correct classification of $X[t + 1]$

Long short-term memory network

- Additional reading:
  - Supervised sequence labelling with recurrent neural networks (Chap. 4)[Graves, 2012]
  - LSTM: A search space odyssey [Greff et al., 2016]
  - Notes on Neural networks (Sec. 7) [Rauber, 2015]
  - The Unreasonable Effectiveness of Recurrent Neural Networks [Karpathy, 2015]
  - Understanding LSTM Networks [Olah, 2015]

Assignment 2: Character-level text generation

Train a long short-term memory network to predict the next character given a sequence of characters. You may use a book available as plain text as a training set (for instance, from Project Guttenberg) or another dataset of your choice.
Use the resulting model to generate text by sampling a character according to the network output and feeding it back as a network input.

Highway/residual layer

- Idea: information should be able to flow across layers unaltered
- Traditional layer:

$$\mathbf{a}^{(l)} = \mathbf{f}(\mathbf{W}^{(l)}\mathbf{a}^{(l-1)} + \mathbf{b}^{(l)})$$

- Residual layer [He et al., 2016]:

$$\mathbf{a}^{(l)} = \mathbf{a}^{(l-1)} + \mathbf{f}(\mathbf{W}^{(l)}\mathbf{a}^{(l-1)} + \mathbf{b}^{(l)})$$

- Highway layer (with coupled gates) [Srivastava et al., 2015]:

$$\mathbf{a}^{(l)} = \mathbf{a}^{(l-1)} \odot \mathbf{g}(\mathbf{a}^{(l-1)}) + \mathbf{f}(\mathbf{W}^{(l)}\mathbf{a}^{(l-1)} + \mathbf{b}^{(l)}) \odot (\mathbf{1} - \mathbf{g}(\mathbf{a}^{(l-1)})),$$

where

$$\mathbf{g}(\mathbf{a}^{(l-1)}) = \boldsymbol{\sigma}(\mathbf{W}^{(l,g)}\mathbf{a}^{(l-1)} + \mathbf{b}^{(l,g)})$$

Sequence to sequence model

- Idea: using an encoding phase followed by a decoding phase to map between sequences of arbitrary lengths [Cho et al., 2014, Sutskever et al., 2014]

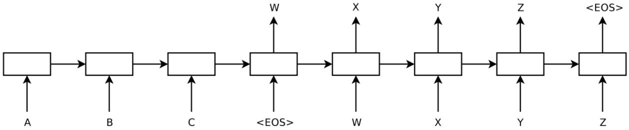


  Image from [Sutskever et al., 2014]

- The recurrent networks that perform encoding and decoding are not necessarily the same

Differentiable neural computer

- Idea: a neural network can learn to read and write from a memory matrix using gating mechanisms [Graves et al., 2016]

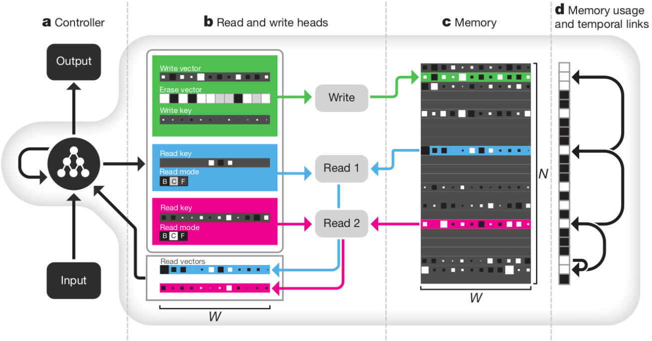


Image from [Graves et al., 2016]

PixelRNN

- Idea: using a recurrent neural network trained to predict each pixel given the previous pixels as a probabilistic model [van den Oord et al., 2016]

$$p(\mathbf{x} \mid \boldsymbol{\theta}) = \prod_{j=1}^{d} p(x_j \mid x_1, \ldots, x_{j-1}, \boldsymbol{\theta})$$

Image from [van den Oord et al., 2016]

Generative adversarial network

- Idea: training a (discriminator) network to discriminate between real and synthetic observations and training another (generator) network to generate synthetic observations from noise that fool the discriminator [Goodfellow et al., 2014]

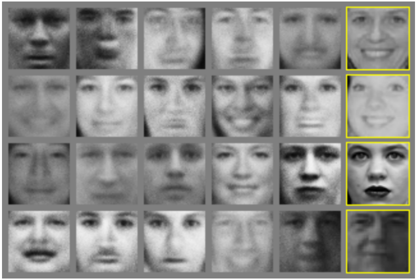


Image from [Goodfellow et al., 2014]

Variational autoencoder

- Idea: training a model with (easy to sample) hidden variables by maximizing a particular lower bound on the log-likelihood [Kingma and Welling, 2014, Rezende et al., 2014]

$$\int_{\text{Val}(\mathbf{Z})} p(\mathbf{x} \mid \mathbf{z}, \boldsymbol{\theta}) p(\mathbf{z} \mid \boldsymbol{\theta})\, d\mathbf{z} = \int_{\text{Val}(\mathbf{Z})} \mathcal{N}(\mathbf{x} \mid \mathbf{f}(\mathbf{z}, \boldsymbol{\theta}), \sigma^2 \mathbf{I}) \mathcal{N}(\mathbf{z} \mid \mathbf{0}, \mathbf{I})\, d\mathbf{z}$$

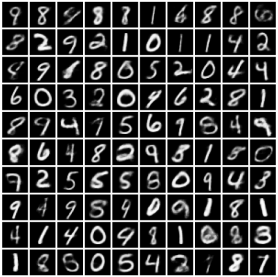


Image from [Doersch, 2016]

Recurrent policy gradient

- Idea: a recurrent neural network represents a policy by a probability distribution over actions given the history of observations and actions [Wierstra et al., 2009]
- The goal is to maximize the expected return J given by

$$J(\boldsymbol{\theta}) = \mathbb{E}\left[\sum_{t=1}^{T} R_t \mid \boldsymbol{\theta}\right] = \sum_{\boldsymbol{\tau}} p(\boldsymbol{\tau} \mid \boldsymbol{\theta}) \sum_{t=1}^{T} r_t,$$

  where $\boldsymbol{\theta}$ are the policy parameters and $\boldsymbol{\tau}$ denotes a trajectory
- It can be shown that $\nabla J(\boldsymbol{\theta})$ is given by

$$\nabla J(\boldsymbol{\theta}) = \mathbb{E}\left[\sum_{t=1}^{T-1} \nabla_{\boldsymbol{\theta}} \log p(A_t \mid X_{1:t}, A_{1:t-1}, \boldsymbol{\theta}) \sum_{t'=t+1}^{T} R_{t'} \mid \boldsymbol{\theta}\right]$$

- A Monte Carlo estimate may be used for gradient ascent

Asynchronous advantage actor-critic

- Idea: asynchronously updating policy parameters shared by several threads using policy gradients with a value-function baseline [Mnih et al., 2016]

Image from [DM1, 2016]

Trust region policy optimization

- Idea: approximating a minorization-maximization procedure that would lead to updates that never deteriorate the policy [Schulman et al., 2015]

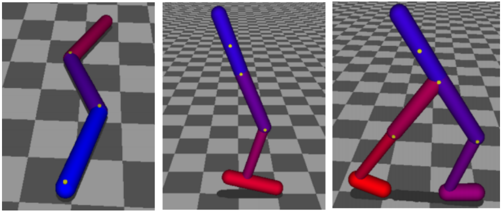


Image from [Schulman et al., 2015]

Separable natural evolution strategies

- We consider a simplified version of separable natural evolution strategies [Wierstra et al., 2014] that was applied to current benchmarks [Salimans et al., 2017]
- Let $J(\boldsymbol{\theta})$ denote the expected return of following a policy parameterized by $\boldsymbol{\theta}$
- Let $p(\boldsymbol{\theta} \mid \boldsymbol{\psi}) = \mathcal{N}(\boldsymbol{\theta} \mid \boldsymbol{\psi}, \sigma^2 \mathbf{I})$ and consider the task of maximizing the expected expected return η given by

$$\eta(\boldsymbol{\psi}) = \mathbb{E}\left[J(\boldsymbol{\Theta}) \mid \boldsymbol{\psi}\right] = \int_{\mathrm{Val}(\boldsymbol{\Theta})} p(\boldsymbol{\theta} \mid \boldsymbol{\psi}) J(\boldsymbol{\theta}) \, d\boldsymbol{\theta}$$

- It can be shown that $\nabla \eta(\boldsymbol{\psi})$ is given by

$$\nabla \eta(\boldsymbol{\psi}) = \sigma^{-1} \mathbb{E}\left[J(\boldsymbol{\psi} + \sigma \boldsymbol{\mathcal{E}}) \boldsymbol{\mathcal{E}}\right], \text{where } \boldsymbol{\mathcal{E}} \sim \mathcal{N}(\cdot \mid \mathbf{0}, \mathbf{I})$$

- A Monte Carlo estimate may be used for gradient ascent

Final assignment

- Implement, improve, or apply a deep learning model [4]
- Prepare and deliver a mid-term presentation (10 minutes) and a final presentation (10 minutes)
- Deliver the resulting code, which should be well organized and documented
- Write a self-contained report including mathematical background, design decisions, and implementation overview. Assume the grade will be based solely on this report

[4] Please consult us about the suitability of your idea. We also have several suggestions, specially for students who are interested in IDSIA

Deep Learning Lab

Paulo Rauber

paulo@idsia.ch

Imanol Schlag

imanol@idsia.ch

November 16, 2017

USI/SUPSI
IDSIA

References I

(2016).
Asynchronous methods for deep reinforcement learning: Labyrinth.
https://www.youtube.com/watch?v=nMR5mjCFZCw.

(2016).
Recurrent neural networks in tensorflow I.
https://r2rt.com/recurrent-neural-networks-in-tensorflow-i.html.

(2017a).
A guide to numpy/scipy documentation.
https://github.com/numpy/numpy/blob/master/doc/HOWTO_DOCUMENT.rst.txt.

(2017).
Institute of computational science HPC.
https://intranet.ics.usi.ch/HPC.

References II

(2017b).
Numpy broadcasting.
https://docs.scipy.org/doc/numpy-dev/user/basics.broadcasting.html.

(2017c).
Numpy quickstart tutorial.
https://docs.scipy.org/doc/numpy-dev/user/quickstart.html.

(2017a).
TensorFlow: Develop.
https://www.tensorflow.org/get_started/.

(2017b).
TensorFlow for deep learning research.
http://web.stanford.edu/class/cs20si/syllabus.html.

References III

(2017c).
TensorFlow: Using gpus.
https://www.tensorflow.org/tutorials/using_gpu.

Abadi, M. et al. (2015).
TensorFlow: Large-scale machine learning on heterogeneous systems.
https://www.tensorflow.org/about/bib.

Bishop, C. M. (2006).
Pattern Recognition and Machine Learning.
Springer.

References IV

Cho, K., van Merrienboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., and Bengio, Y. (2014).
Learning phrase representations using rnn encoder–decoder for statistical machine translation.
In Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP).

Doersch, C. (2016).
Tutorial on variational autoencoders.
arXiv preprint arXiv:1606.05908.

Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., and Bengio, Y. (2014).
Generative adversarial nets.
In Advances in neural information processing systems.

References V

Graves, A. (2012).
Supervised sequence labelling with recurrent neural networks.
Springer.

Graves, A., Wayne, G., Reynolds, M., Harley, T., Danihelka, I., Grabska-Barwińska, A., Colmenarejo, S. G., Grefenstette, E., Ramalho, T., Agapiou, J., et al. (2016).
Hybrid computing using a neural network with dynamic external memory.
Nature.

Greff, K., Srivastava, R. K., Koutník, J., Steunebrink, B. R., and Schmidhuber, J. (2016).
LSTM: A search space odyssey.
IEEE transactions on neural networks and learning systems.

References VI

He, K., Zhang, X., Ren, S., and Sun, J. (2016).
Deep residual learning for image recognition.
In Proceedings of the IEEE conference on computer vision and pattern recognition.

Karpathy, A. (2015).
The unreasonable effectiveness of recurrent neural networks.
http://karpathy.github.io/2015/05/21/rnn-effectiveness/.

Kingma, D. P. and Welling, M. (2014).
Auto-encoding variational bayes.
In International Conference on Learning Representations.

Li, F.-F. and Karpathy, A. (2015).
Convolutional neural networks for visual recognition.
http://cs231n.github.io/convolutional-networks.

References VII

Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T., Harley, T., Silver, D., and Kavukcuoglu, K. (2016).
Asynchronous methods for deep reinforcement learning.
In International Conference on Machine Learning.

Murphy, K. P. (2012).
Machine learning: a probabilistic perspective.
MIT Press.

Nielsen, M. (2015).
Neural networks and deep learning.
Determination Press.
http://neuralnetworksanddeeplearning.com.

References VIII

Olah, C. (2015).
Understanding LSTM networks.
http://colah.github.io/posts/2015-08-Understanding-LSTMs/.

Pilgrim, M. (2011).
Dive into Python 3.
http://www.diveintopython3.net/.

Rauber, P. E. (2015).
Notes on neural networks.
http://paulorauber.com/notes/neural_networks.pdf.

Rauber, P. E. (2016).
Notes on machine learning.
http://paulorauber.com/notes/machine_learning.pdf.

References IX

- Rezende, D. J., Mohamed, S., and Wierstra, D. (2014).
Stochastic backpropagation and approximate inference in deep generative models.
In International Conference on Machine Learning.

- Rossum, G. v., Warsaw, B., and Coghlan, N. (2001).
PEP 8 – style guide for Python code.
https://www.python.org/dev/peps/pep-0008/.

- Ruder, S. (2016).
An overview of gradient descent optimization algorithms.
http://ruder.io/optimizing-gradient-descent/.

- Salimans, T., Ho, J., Chen, X., and Sutskever, I. (2017).
Evolution strategies as a scalable alternative to reinforcement learning.
arXiv preprint arXiv:1703.03864.

Schulman, J., Levine, S., Abbeel, P., Jordan, M., and Moritz, P. (2015).
Trust region policy optimization.
In International Conference on Machine Learning.

Srivastava, R. K., Greff, K., and Schmidhuber, J. (2015).
Training very deep networks.
In Advances in neural information processing systems.

Sutskever, I., Vinyals, O., and Le, Q. V. (2014).
Sequence to sequence learning with neural networks.
In Advances in neural information processing systems.

van den Oord, A., Kalchbrenner, N., Espeholt, L., Vinyals, O., Graves, A., et al. (2016).
Conditional image generation with pixelCNN decoders.
In Advances in Neural Information Processing Systems.

References XI

Wierstra, D., Förster, A., Peters, J., and Schmidhuber, J. (2009).
Recurrent policy gradients.
Logic Journal of IGPL, 18(5).

Wierstra, D., Schaul, T., Glasmachers, T., Sun, Y., Peters, J., and Schmidhuber, J. (2014).
Natural evolution strategies.
Journal of Machine Learning Research.