Deep Learning

Jianlin Cheng
Department of EECS
University of Missouri - Columbia
Acknowledgements

Some content of this teaching presentation was drawn from many sources created by great scientists in the field of deep learning.
**Simplest Neural Network for Classification — Logistic Regression**

- **Binary Classification**

**Activation**

activation = \( w_0 + w_1x_1 + w_2x_2 + \ldots, w_dx_d \)

**Probability**

\[ P(y = 1) = f(x) = \frac{1}{1 + e^{-w_0 - \sum w_i x_i}} \]

**Loss/cross-entropy**

\[ -(y \log(o) + (1-y) \log(1-o)) \]

\[ O: \text{output } f(x) \]
\[ y: \text{target (0 or 1)} \]
\[ g: \text{sigmoid function} \]
\[ a: \text{activation} \]

**Update Rule**

\[ w^{new} = w^{cur} - \frac{\partial \text{Error}}{\partial w} = w^{cur} - (f(x) - y)x \]
Simplest Neural Network for Classification – Logistic Regression

• Multi Classification

 activation \( = w_0 + w_1x_1 + w_2x_2 + \ldots, w_dx_d \)

\[ P(y_i = 1) = f(x) = \frac{e^{w_{i0} + \sum w_{ij}x_j}}{\sum e^{w_{i0} + \sum w_{ij}x_j}} \]

Loss/cross-entropy: \[ -\sum_{i=1}^{m} y_i \log o_i \]

O: output \( f(x) \)

y: target (0 or 1)

g: softmax

a: activation
Perceptron

Learning to map input to output (label) and is guided by output.

Training

\[ f(x) = w_0 + w_1 x_1 + w_2 x_2 + \ldots, w_d x_d \]

Error: \((o - y)^2\)

\(f(x) > 0\): positive class (1)

\(f(x) < 0\): negative class (0)

Activation function: Delta function

Learning is to adjust \(w\) to minimize the squared error between \(f(x)\) and true \(y\).

Testing

\[ w_{new} = w_{cur} - \frac{\partial Error}{\partial w} = w_{cur} - (o - y)x \]

Perceptron – 1960s
Simplest Neural Network for Regression – Linear Regression

Output = $w_0 + w_1 x_1 + w_2 x_2 + \ldots, w_d x_d$

$w^{new} = w^{cur} - \frac{\partial Error}{\partial w} = w^{cur} - (o - y)x$

Error: $(o-y)^2$

Y: target
O: output
Activation function: identity function
a: activation

$I(a)$

$w_0 \quad w_1 \quad w_2 \quad \ldots \quad w_d$

$I \quad x_1 \quad x_2 \quad \ldots \quad x_d$
TensorFlow Demo of One-Node Network

• Data Sets: MNIST digit recognition data or Iris flower classification data
• Google’s TensorFlow installation: https://www.tensorflow.org/
• Install it on mac: https://www.tensorflow.org/install/install_mac
• Activate tensorflow: $source ~/tensorflow/bin/activate
```python
import tensorflow as tf

# Model parameters
W = tf.Variable([.3], dtype=tf.float32)
b = tf.Variable([- .3], dtype=tf.float32)

# Model input and output
x = tf.placeholder(tf.float32)
linear_model = W * x + b
y = tf.placeholder(tf.float32)

# loss
loss = tf.reduce_sum(tf.square(linear_model - y))  # sum of the squares

# optimizer
optimizer = tf.train.GradientDescentOptimizer(0.01)
train = optimizer.minimize(loss)

# training data
x_train = [1, 2, 3, 4]
y_train = [0, -1, -2, -3]

# training loop
init = tf.global_variables_initializer()
sess = tf.Session()
sess.run(init)  # reset values to wrong
for i in range(1000):
    sess.run(train, {x: x_train, y: y_train})

# evaluate training accuracy
curr_W, curr_b, curr_loss = sess.run([W, b, loss], {x: x_train, y: y_train})
print("W: \%s b: \%s loss: \%s\"%(curr_W, curr_b, curr_loss))
```
import tensorflow as tf

# NumPy is often used to load, manipulate and preprocess data.
import numpy as np

# Declare list of features. We only have one numeric feature. There are many
# other types of columns that are more complicated and useful.
feature_columns = [tf.feature_column.numeric_column("x", shape=[1])]

# An estimator is the front end to invoke training (fitting) and evaluation
# (inference). There are many predefined types like linear regression,
# linear classification, and many neural network classifiers and regressors.
# The following code provides an estimator that does linear regression.
estimator = tf.estimator.LinearRegressor(feature_columns=feature_columns)

# TensorFlow provides many helper methods to read and set up data sets.
# Here we use two data sets: one for training and one for evaluation
# We have to tell the function how many batches
# of data (num_epochs) we want and how big each batch should be.
x_train = np.array([1., 2., 3., 4.])
y_train = np.array([0., -1., -2., -3.])
x_eval = np.array([2., 5., 8., 1.])
y_eval = np.array([-1.01, -4.1, -7, 0.])
input_fn = tf.estimator.inputs.numpy_input_fn(
    {'x': x_train}, y_train, batch_size=4, num_epochs=None, shuffle=True)
train_input_fn = tf.estimator.inputs.numpy_input_fn(
    {'x': x_train}, y_train, batch_size=4, num_epochs=1000, shuffle=False)
eval_input_fn = tf.estimator.inputs.numpy_input_fn(
    {'x': x_eval}, y_eval, batch_size=4, num_epochs=1000, shuffle=False)

# We can invoke 1000 training steps by invoking the method and passing the
# training data set.
estimator.train(input_fn=input_fn, steps=1000)

# Here we evaluate how well our model did.
train_metrics = estimator.evaluate(input_fn=train_input_fn)
eval_metrics = estimator.evaluate(input_fn=eval_input_fn)
print("train metrics: %r"% train_metrics)
print("eval metrics: %r"% eval_metrics)
Logistic Regression for Hand Writing Classification

Each image is 28 pixels by 28 pixels. We can interpret this as a big array of numbers:

We can flatten this array into a vector of $28 \times 28 = 784$ numbers. It doesn't matter how we flatten the array, as long as we're consistent between images. From this perspective, the MNIST images are just a bunch of points in a 784-dimensional vector space, with a very rich structure (warning: computationally intensive visualizations).
If we write that out as equations, we get:

$$
\begin{bmatrix}
  y_1 \\
  y_2 \\
  y_3
\end{bmatrix} = \text{softmax}
\begin{bmatrix}
  W_{1,1} x_1 + W_{1,2} x_2 + W_{1,3} x_3 + b_1 \\
  W_{2,1} x_1 + W_{2,2} x_2 + W_{2,3} x_3 + b_2 \\
  W_{3,1} x_1 + W_{3,2} x_2 + W_{3,3} x_3 + b_3
\end{bmatrix}
$$

We can "vectorize" this procedure, turning it into a matrix multiplication and vector addition. This is helpful for computational efficiency. (It's also a useful way to think.)

$$
\begin{bmatrix}
  y_1 \\
  y_2 \\
  y_3
\end{bmatrix} = \text{softmax}
\begin{bmatrix}
  W_{1,1} & W_{1,2} & W_{1,3} \\
  W_{2,1} & W_{2,2} & W_{2,3} \\
  W_{3,1} & W_{3,2} & W_{3,3}
\end{bmatrix}
\begin{bmatrix}
  x_1 \\
  x_2 \\
  x_3
\end{bmatrix} +
\begin{bmatrix}
  b_1 \\
  b_2 \\
  b_3
\end{bmatrix}
$$

More compactly, we can just write:

$$
y = \text{softmax}(W x + b)
$$
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

import argparse
import sys

from tensorflow.examples.tutorials.mnist import input_data

import tensorflow as tf

FLAGS = None

def main(_):
  # Import data
  mnist = input_data.read_data_sets(FLAGS.data_dir, one_hot=True)

  # Create the model
  x = tf.placeholder(tf.float32, [None, 784])
  W = tf.Variable(tf.zeros([784, 10]))
  b = tf.Variable(tf.zeros([10]))
  y = tf.matmul(x, W) + b

  # Define loss and optimizer
  y_ = tf.placeholder(tf.float32, [None, 10])

  # The raw formulation of cross-entropy,
  #
```python
# tf.reduce_mean(-tf.reduce_sum(y_ * tf.log(tf.nn.softmax(y)),
#                            reduction_indices=[1]))
#
# # can be numerically unstable.
# #
# # So here we use tf.nn.softmax_cross_entropy_with_logits on the raw
# # outputs of 'y', and then average across the batch.
# cross_entropy = tf.reduce_mean(
#     tf.nn.softmax_cross_entropy_with_logits(labels=y_, logits=y))
# train_step = tf.train.GradientDescentOptimizer(0.5).minimize(cross_entropy)
#
# sess = tf.InteractiveSession()
# tf.global_variables_initializer().run()
# # Train
# for _ in range(1000):
#     batch_xs, batch_ys = mnist.train.next_batch(100)
#     sess.run(train_step, feed_dict={x: batch_xs, y_: batch_ys})
#
# # Test trained model
# correct_prediction = tf.equal(tf.argmax(y, 1), tf.argmax(y_, 1))
# accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
# print(sess.run(accuracy, feed_dict={x: mnist.test.images,
#                                      y_: mnist.test.labels}))
#
# if __name__ == '__main__':
#     parser = argparse.ArgumentParser()
#     parser.add_argument('--data_dir', type=str, default='/tmp/tensorflow/mnist/input_data',
#                         help='Directory for storing input data')
#     FLAGS, unparsed = parser.parse_known_args()
#     tf.app.run(main=main, argv=[sys.argv[0]] + unparsed)
```
Neural Network

Forward Propagation

Hidden layer

Backward Propagation

Error: \((o-y)^2\)

1980s – Neural Network Revolution
Two-Layer Neural Network

Output

Activation function: \( f \) (linear, sigmoid, softmax)

Activation of unit \( a_k \):

\[
\sum_{j=0}^{M} w_{kj} z_j
\]

Activation function: \( g \) (linear, tanh, sigmoid)

Activation of unit \( a_j \):

\[
\sum_{i=0}^{d} w_{ji} x_i
\]

\[
y_k = f\left(\sum_{j=0}^{M} w_{kj} z_j \times g\left(\sum_{i=0}^{d} w_{ji} x_i\right)\right)
\]
 Adjust Weights by Training

- How to adjust weights?
- Adjust weights using known examples (training data) \((x_1, x_2, x_3, \ldots, x_d, t)\).
- Try to adjust weights so that the difference between the output of the neural network \(y\) and \(t\) (target) becomes smaller and smaller.
- Goal is to minimize Error (difference) as we did for one layer neural network.
Adjust Weights using Gradient Descent

Known:
Data: \((x_1, x_2, x_3, ..., x_n)\) target \(t\).

Unknown weights \(w\):
\(w_{11}, w_{12}, \ldots\)

Randomly initialize weights
Repeat
for each example, compute output \(y\) calculate error \(E = (y-t)^2\)
compute the derivative of \(E\) over \(w\):
\(dw = \frac{\partial E}{\partial w}\)
\(w_{\text{new}} = w_{\text{prev}} - \eta \times dw\)

Until error doesn’t decrease or max num of iterations (epochs)

Note: \(\eta\) is learning rate or step size.
Stochastic Gradient Descent

Known:
- Data: \((x_1, x_2, x_3, \ldots, x_n)\) target \(t\).

Unknown weights \(w\):
- \(w_{11}, w_{12}, \ldots\)

Randomly initialize weights

Repeat
- Randomize the order of examples, divide into batches
- For each example in a batch, compute output \(y\) and error \(E = (y-t)^2\)
- Compute the derivative of \(E\) over \(w\): \(dw = \frac{\partial E}{\partial w}\)
- Add the derivatives of a batch together
- \(w_{\text{new}} = w_{\text{prev}} - \eta \ast dw\)

Until error doesn’t decrease or max num of iterations (epochs)

Note: \(\eta\) is learning rate or step size.
Neural Network Learning: Two Processes

• Forward propagation: present an example (data) into neural network. Compute activation into units and output from units.

• Backward propagation: propagate error back from output layer to the input layer and compute derivatives (or gradients).
Forward Propagation

Output
- Activation function: $f$ (linear, sigmoid, softmax)
- Activation of unit $a_k$:
  \[ \sum_{j=1}^{M} w_{kj} z_j \]
- Activation of unit $a_j$:
  \[ \sum_{i=1}^{d} w_{ji} x_i \]

Time complexity?
Forward Propagation

Output
Activation function: \( f \) (linear, sigmoid, softmax)

Activation of unit \( a_k \):
\[
\sum_{j=1}^{M} w_{kj} z_j
\]

Activation function: \( g \) (linear, tanh, sigmoid)

Activation of unit \( a_j \):
\[
\sum_{i=1}^{d} w_{ji} x_i
\]

Time complexity?
\( O(dM + MC) = O(W) \)
Backward Propagation

\[ E = \frac{1}{2} \sum_{k=1}^{C} (y_k - t_k)^2 \]

\[ \frac{\partial E}{\partial y_k} = y_k - t_k \]

\[ a_k = \sum_{j=1}^{M} w_{kj} z_j \]

\[ \frac{\partial E}{\partial a_k} = \frac{\partial E}{\partial y_k} \frac{\partial y_k}{\partial a_k} = (y_k - t_k) f'(a_k) = \delta_k \]

\[ \frac{\partial E}{\partial w_{kj}} = \frac{\partial E}{\partial a_k} \frac{\partial a_k}{\partial w_{kj}} = \delta_k z_j \]

\[ g \sum_{k=1}^{C} \frac{\partial E}{\partial y_k} \frac{\partial y_k}{\partial a_k} \frac{\partial a_k}{\partial z_j} \frac{\partial z_j}{\partial a_j} = \sum_{k=1}^{C} \delta_k w_{kj} g'(a_j) = \delta_j \]

\[ \frac{\partial E}{\partial w_{ji}} = \frac{\partial E}{\partial a_j} \frac{\partial a_j}{\partial w_{ji}} = \delta_j x_i \]

Time complexity?
If no back-propagation, time complexity is: (MdC+CM)
Example

\[ E = \frac{1}{2} (y - t)^2 \]

\[ \delta = \frac{\partial E}{\partial a_k} = \frac{\partial E}{\partial y} \frac{\partial y}{\partial a_k} = (y - t) \]

**f linear function**

\[ a_k: \]

**g is sigmoid:**

\[ \frac{\partial E}{\partial w_j} = \delta z_j \]

\[ \delta_j = \delta w_j g'(a_j) = (y - t)w_j z_j (1 - z_j) \]

\[ \frac{\partial E}{\partial w_{ji}} = \delta_j x_i = (y - t)w_j z_j (1 - z_j) x_i \]
TensorFlow Demo of Simple Neural Networks

• Data Sets: Iris flower classification data
• Google’s TensorFlow installation: https://www.tensorflow.org/

From left to right, *Iris setosa* (by Radomil, CC BY-SA 3.0), *Iris versicolor* (by Dlanglois, CC BY-SA 3.0), and *Iris virginica* (by Frank Mayfield, CC BY-SA 2.0).
<table>
<thead>
<tr>
<th>Sepal Length</th>
<th>Sepal Width</th>
<th>Petal Length</th>
<th>Petal Width</th>
<th>Species</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>3.5</td>
<td>1.4</td>
<td>0.2</td>
<td>0</td>
</tr>
<tr>
<td>4.9</td>
<td>3.0</td>
<td>1.4</td>
<td>0.2</td>
<td>0</td>
</tr>
<tr>
<td>4.7</td>
<td>3.2</td>
<td>1.3</td>
<td>0.2</td>
<td>0</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>7.0</td>
<td>3.2</td>
<td>4.7</td>
<td>1.4</td>
<td>1</td>
</tr>
<tr>
<td>6.4</td>
<td>3.2</td>
<td>4.5</td>
<td>1.5</td>
<td>1</td>
</tr>
<tr>
<td>6.9</td>
<td>3.1</td>
<td>4.9</td>
<td>1.5</td>
<td>1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>6.5</td>
<td>3.0</td>
<td>5.2</td>
<td>2.0</td>
<td>2</td>
</tr>
<tr>
<td>6.2</td>
<td>3.4</td>
<td>5.4</td>
<td>2.3</td>
<td>2</td>
</tr>
<tr>
<td>5.9</td>
<td>3.0</td>
<td>5.1</td>
<td>1.8</td>
<td>2</td>
</tr>
</tbody>
</table>

For this tutorial, the Iris data has been randomized and split into two separate CSVs:

- A training set of 120 samples (iris_training.csv)
- A test set of 30 samples (iris_test.csv).
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

import os
import urllib

import numpy as np
import tensorflow as tf

# Data sets
IRIS_TRAINING = "iris_training.csv"
IRIS_TRAINING_URL = "http://download.tensorflow.org/data/iris_training.csv"

IRIS_TEST = "iris_test.csv"
IRIS_TEST_URL = "http://download.tensorflow.org/data/iris_test.csv"

def main():
    # If the training and test sets aren't stored locally, download them.
    if not os.path.exists(IRIS_TRAINING):
        raw = urllib.urlopen(IRIS_TRAINING_URL).read()
        with open(IRIS_TRAINING, "w") as f:
            f.write(raw)

    if not os.path.exists(IRIS_TEST):
        raw = urllib.urlopen(IRIS_TEST_URL).read()
        with open(IRIS_TEST, "w") as f:
            f.write(raw)

    # Load datasets.
    training_set = tf.contrib.learn.datasets.base.load_csv_with_header(
        filename=IRIS_TRAINING,
        target_dtype=np.int,
        features_dtypes=np.float32)
    test_set = tf.contrib.learn.datasets.base.load_csv_with_header(
        filename=IRIS_TEST,
        target_dtype=np.int,
        features_dtypes=np.float32)
# Specify that all features have real-value data
defeature_columns = [tf.feature_column.numeric_column("x", shape=[4])]

# Build 3 layer DNN with 10, 20, 10 units respectively.
classifier = tf.estimator.DNNClassifier(  
    feature_columns=feature_columns,  
    hidden_units=[10, 20, 10],  
    n_classes=3,  
    model_dir="/tmp/iris_model")

# Define the training inputs
train_input_fn = tf.estimator.inputs.numpy_input_fn(  
    x={"x": np.array(training_set.data)},  
    y=np.array(training_set.target),  
    num_epochs=None,  
    shuffle=True)

# Train model.
classifier.train(input_fn=train_input_fn, steps=2000)

# Define the test inputs
test_input_fn = tf.estimator.inputs.numpy_input_fn(  
    x={"x": np.array(test_set.data)},  
    y=np.array(test_set.target),  
    num_epochs=1,  
    shuffle=False)

# Evaluate accuracy.
accuracy_score = classifier.evaluate(input_fn=test_input_fn)["accuracy"]

print("\nTest Accuracy: {0:0.3f}".format(accuracy_score))

# Classify two new flower samples.
new_samples = np.array([[[6.4, 3.2, 4.5, 1.5],  
                         [5.8, 3.1, 5.0, 1.7]],  
                         dtype=np.float32])
predict_input_fn = tf.estimator.inputs.numpy_input_fn(  
    x={"x": new_samples},  
    num_epochs=1,  
    shuffle=False)
predictions = list(classifier.predict(input_fn=predict_input_fn))
predicted_classes = [p['classes'] for p in predictions]

print("New Samples, Class Predictions: {}\n".format(predicted_classes))

if __name__ == '__main__':
    main()
Vanishing Gradient or Explosion
Neural Network’s Winter in 1990s

• A standard three-layer neural network is a universal approximator

• Hard to train multi-layer neural networks

• Get different models from different training (local minimal)

\[ y_k = f \left( \sum_{j=0}^{M} w_{kj} \times g \left( \sum_{i=0}^{d} w_{ji} x_i \right) \right) \]
How to Construct Deep Networks?

G. Hinton

2000s
Learning by Composition? – A Face Recognition Analogy

Face or not?

Lines, circles, squares

Image pixels

Brain Learning
Breakthrough

Deep Learning: machine learning algorithms based on learning multiple levels of representation / abstraction

Amazing improvements in error rate in object recognition, object detection, speech recognition, and more recently, in natural language processing / understanding
Machine Learning for Artificial Intelligence

Four key ingredients for ML towards AI

- Lots & lots of data
- Very flexible models
- Enough computing power
- Powerful priors that can defeat the curse of dimensionality
Bypassing the curse of dimensionality

• We need to build compositionality into our ML models just as human languages exploit compositionality

• Exploiting compositionality gives an exponential gain in representational power: (1) distributed representations/embeddings (feature learning); (2) deep architecture (multi-levels of feature learning)

• Additional prior: compositionality is useful to describe the world around us efficiently
Classical Symbolic AI vs Learning Distributed Representations

- Two symbols are equally far from each other
- Concepts are not represented by symbols in our brain, but by patterns of activation

(Connectionism, 1980's)
Exponential advantage of distributed representations

Learning a set of parametric features that are not mutually exclusive can be exponentially more statistically efficient than having nearest-neighbor-like or clustering-like models
Each feature can be discovered without the need for seeing the exponentially large number of configurations of the other features

- Consider a network whose hidden units discover the following features:
  - Person wears glasses
  - Person is female
  - Person is a child
  - Etc.

If each of $n$ feature requires $O(k)$ parameters, need $O(nk)$ examples

Non-parametric methods would require $O(n^d)$ examples
Exponential advantage of distributed representations

Prop. 2 of *Pascanu, Montufar & Bengio ICLR’2014*: number of pieces distinguished by 1-hidden-layer rectifier net with $n$ units and $d$ inputs (i.e. $O(nd)$ parameters) is

$$\sum_{j=0}^{d} \binom{n}{j} = O(n^d)$$
Deep Learning: Automating Feature Discovery

- Input
  - Rule-based systems
  - Classic machine learning
  - Representation learning
  - Deep learning

- Mapping from features
  - Most complex features
  - Simplest features

- Output
  - Mapped from features
Exponential advantage of depth

Theoretical arguments:

- Logic gates
- Formal neurons
- RBF units

2 layers of RBMs & auto-encoders = universal approximator

Theorems on advantage of depth:

Some functions compactly represented with \( k \) layers may require exponential size with 2 layers
Why does it work? No Free Lunch

- It only works because we are making some assumptions about the data generating distribution
- Worse-case distributions still require exponential data
- But the world has structure and we can get an exponential gain by exploiting some of it
Exponential advantage of depth

- Expressiveness of deep networks with piecewise linear activation functions: exponential advantage for depth (Montufar et al, NIPS 2014)

- Number of pieces distinguished for a network with depth $L$ and $n_i$ units per layer is at least

$$\left( \prod_{i=1}^{L-1} \left[ \frac{n_i}{n_0} \right]^{n_0} \right) \sum_{j=0}^{n_0} \binom{n_L}{j}$$

or, if hidden layers have width $n$ and input has size $n_0$

$$\Omega \left( \left( \frac{n}{n_0} \right)^{(L-1)n_0} nn_0 \right)$$
Construct Deep Networks

Backprop
(modular approach)
Typical Deep Multilayer Neural Net

- Complex learning machines can be built by assembling modules into networks
- Linear Module
  \[ \text{Out} = W \cdot \text{In} + B \]
- ReLU Module (Rectified Linear Unit)
  \[ \text{Out}_i = 0 \quad \text{if } \text{In}_i < 0 \]
  \[ \text{Out}_i = \text{In}_i \quad \text{otherwise} \]
- Cost Module: Squared Distance
  \[ C = ||\text{In}_1 - \text{In}_2||^2 \]
- Objective Function
  \[ L(\Theta) = \frac{1}{p} \sum_k C(X^k, Y^k, \Theta) \]
  \[ \Theta = (W1, B1, W2, B2, W3, B3) \]

X (input) \[ \rightarrow \] \[ \text{Squared Distance} \] \[ \rightarrow \] \[ \text{Y (desired output)} \]
All major deep learning frameworks use modules (inspired by SN/Lush, 1991)

- Torch7, Theano, TensorFlow...

```
-- sizes
ninput = 28*28  -- e.g. for MNIST
nhidden1 = 1000
noutput = 10

-- network module
net = nn.Sequential()
net:add(nn.Linear(ninput, nhidden))
net:add(nn.Threshold())
net:add(nn.Linear(nhidden, noutput))
net:add(nn.LogSoftMax())

-- cost module
cost = nn.ClassNLLCriterion()

-- get a training sample
input = trainingset.data[k]
target = trainingset.labels[k]

-- run through the model
output = net:forward(input)
c = cost:forward(output, target)
```
Computing Gradients by Back-Propagation

- A practical Application of Chain Rule
- Backprop for the state gradients:
  - \( \frac{dC}{dx_{i-1}} = \frac{dC}{dx_i} \cdot \frac{dx_i}{dx_{i-1}} \)
  - \( \frac{dC}{dx_{i-1}} = \frac{dC}{dx_i} \cdot \frac{dF_i(x_{i-1}, w_i)}{dx_{i-1}} \)
- Backprop for the weight gradients:
  - \( \frac{dC}{dw_i} = \frac{dC}{dx_i} \cdot \frac{dx_i}{dw_i} \)
  - \( \frac{dC}{dw_i} = \frac{dC}{dx_i} \cdot \frac{dF_i(x_{i-1}, w_i)}{dw_i} \)
Running Backprop

- Torch7 example
- Gradtheta contains the gradient

\[ C(X, Y, \Theta) \]

```
-- network module
net = nn.Sequential()
net:add(nn.Linear(ninput, nhidden))
net:add(nn.Threshold())
net:add(nn.Linear(nhidden, noutput))
net:add(nn.LogSoftMax())

-- cost module
cost = nn.ClassNLLCriterion()

-- gather the parameters in a vector
theta, gradtheta = net:getParameters()

-- get a training sample
input = trainingset.data[k]
target = trainingset.labels[k]

-- run through the model
output = net:forward(input)
c = cost:forward(output, target)

-- run backprop
gradtheta:zero()
gradoutput = cost:backward(output, target)
net:backward(input, gradoutput)
```
Modular Classes

**Linear**
- \( Y = W \cdot X \); \( \frac{dC}{dX} = W^T \cdot \frac{dC}{dY} \); \( \frac{dC}{dW} = \frac{dC}{dY} \cdot \frac{dY}{dx} \)

**ReLU**
- \( y = \text{ReLU}(x) \); if \( x < 0 \) \( \frac{dC}{dx} = 0 \) else \( \frac{dC}{dx} = \frac{dC}{dy} \)

**Duplicate**
- \( Y_1 = X, Y_2 = X \); \( \frac{dC}{dX} = \frac{dC}{dY_1} + \frac{dC}{dY_2} \)

**Add**
- \( Y = X_1 + X_2 \); \( \frac{dC}{dX_1} = \frac{dC}{dY} \); \( \frac{dC}{dX_2} = \frac{dC}{dY} \)

**Max**
- \( y = \max(x_1, x_2) \); if \( x_1 > x_2 \) \( \frac{dC}{dx_1} = \frac{dC}{dy} \) else \( \frac{dC}{dx_1} = 0 \)

**LogSoftMax**
- \( Y_i = X_i - \log \left[ \sum_j \exp(X_j) \right] \); ...
Modular Classes

- Many more basic module classes
- Cost functions:
  - Squared error
  - Hinge loss
  - Ranking loss
- Non-linearities and operators
  - ReLU, “leaky” ReLU, abs,....
  - Tanh, logistic
  - Just about any simple function (log, exp, add, mul,....)
- Specialized modules
  - Multiple convolutions (1D, 2D, 3D)
  - Pooling/subsampling: max, average, Lp, log(sum(exp())), maxout
  - Long Short-Term Memory, attention, 3-way multiplicative interactions.
  - Switches
  - Normalizations: batch norm, contrast norm, feature norm...
  - inception (replace linear filter with non-linear filter in convolutional neural network)
Hinge loss

From Wikipedia, the free encyclopedia

In machine learning, the **hinge loss** is a loss function used for training classifiers. The hinge loss is used for "maximum-margin" classification, most notably for support vector machines (SVMs).\(^1\) For an intended output \(t = \pm 1\) and a classifier score \(y\), the hinge loss of the prediction \(y\) is defined as

\[
\ell(y) = \max(0, 1 - t \cdot y)
\]

Note that \(y\) should be the "raw" output of the classifier's decision function, not the predicted class label. For instance, in linear SVMs, \(y = w \cdot x + b\), where \((w, b)\) are the parameters of the hyperplane and \(x\) is the point to classify.

It can be seen that when \(t\) and \(y\) have the same sign (meaning \(y\) predicts the right class) and \(|y| \geq 1\), the hinge loss \(\ell(y) = 0\), but when they have opposite sign, \(\ell(y)\) increases linearly with \(y\) (one-sided error).
Plot of hinge loss (blue, measured vertically) vs. zero-one loss (measured vertically; misclassification, green: $y < 0$) for $t = 1$ and variable $y$ (measured horizontally). Note that the hinge loss penalizes predictions $y < 1$, corresponding to the notion of a margin in a support vector machine.
Any architecture works

- Any connection graph is permissible
  - Directed acyclic graphs (DAG)
  - Networks with loops must be “unfolded in time”.

- Any module is permissible
  - As long as it is continuous and differentiable almost everywhere with respect to the parameters, and with respect to non-terminal inputs.

- Most frameworks provide automatic differentiation
  - Theano, Torch7+autograd,…
  - Programs are turned into computation DAGs and automatically differentiated.
Backprop in Practice

- Use ReLU non-linearities
- Use cross-entropy loss for classification
- Use Stochastic Gradient Descent on minibatches
- Shuffle the training samples (← very important)
- Normalize the input variables (zero mean, unit variance)
- Schedule to decrease the learning rate
- Use a bit of L1 or L2 regularization on the weights (or a combination)
  - But it's best to turn it on after a couple of epochs
- Use “dropout” for regularization
- Lots more in [LeCun et al. “Efficient Backprop” 1998]
Convolutional Networks
Deep Learning = Training Multistage Machines

- **Traditional Pattern Recognition**: Fixed/Handcrafted Feature Extractor
- **Mainstream Pattern Recognition**: Until recently
- **Deep Learning**: Multiple stages/layers trained end to end
Overall Architecture: multiple stages of Normalization → Filter Bank → Non-Linearity → Pooling

- **Normalization**: variation on whitening (optional)
  - Subtractive: average removal, high pass filtering
  - Divisive: local contrast normalization, variance normalization

- **Filter Bank**: dimension expansion, projection on overcomplete basis

- **Non-Linearity**: sparsification, saturation, lateral inhibition...
  - Rectification (ReLU), tanh, ...

- **Pooling**: aggregation over space or feature type
  - Max, log prob.
Convolutional Architecture

Filter Bank +non-linearity
Pooling
Filter Bank +non-linearity
Pooling
Filter Bank +non-linearity

LeNet1 [LeCun et al. NIPS 1989]
Multiple Convolutions

Input Volume (+pad 1) (7x7x3)

Filter W0 (3x3x3)

Filter W1 (3x3x3)

Output Volume (3x3x2)

Convolutional Networks (vintage 1990)

filters → tanh → average-tanh → filters → tanh → average-tanh → filters → tanh
1D (Temporal) convolutional net

- 1D (Temporal) ConvNet, aka Timed-Delay Neural Nets
- Groups of units are replicated at each time step.
- Replicas have identical (shared) weights.
1D CNN for Protein Fold Classification
Deep 1D-Convolutional Neural Network

INPUT
L×45

Conv Layer 1
10×(L×2)

Conv Layer 2
10×(L×2)

... ...

Conv Layer 10
10×(L×2)

Pooling Layer
10×(30×2)

Flatten Layer
(1×600)

Dense layer
(1×500)

Output
(1×1195)

Rectified Linear Unit (ReLU): \( f(x) = \max(0, x) \)
Output layer: 1,195 nodes with sigmoid function
Training Data

Distribution of sequence lengths of protein domains in SCOP 1.75 database

16,712 protein domains
Batch Training Using Binning and Padding according to Sequence Length
Demo of Training DCNN

Protein Sequence
ASCETTVTSGDTMYSRSIVPASCAETFVNFEEHKHMPKTGMGHNVWLAKSADVGDVAKECAHAGADNNFVTGDKRVIAFTPIGGGEKTSVKFKVSAKLSDKDEAYTTFCSYPGHFSMMLRTKLLE

1195 Folds in SCOP 1.75

Native Structure (Fold b.6)
## Accuracy on Validation Datasets of SCOP 1.75

<table>
<thead>
<tr>
<th>Num of Predictions</th>
<th>Set 1 (Sim &lt; 95%)</th>
<th>Set 2 (Sim &lt; 70%)</th>
<th>Set 3 (Sim &lt; 40%)</th>
<th>Set 4 (Sim &lt; 25%)</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>Top 1</td>
<td>80.4%</td>
<td>78.2%</td>
<td>75.8%</td>
<td>67.0%</td>
<td>75.3%</td>
</tr>
<tr>
<td>Top 5</td>
<td>93.7%</td>
<td>92.4%</td>
<td>90.0%</td>
<td>87.6%</td>
<td>91.0%</td>
</tr>
</tbody>
</table>

## Accuracy on Independent Dataset of SCOP 2.06 (4,418 proteins, Sim <= 40%)

<table>
<thead>
<tr>
<th>Method</th>
<th>Top 1</th>
<th>Top 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>DeepSF</td>
<td>77%</td>
<td>92%</td>
</tr>
<tr>
<td>MajorityAssignment</td>
<td>4%</td>
<td>16%</td>
</tr>
</tbody>
</table>
Simple ConvNet for MNIST [LeCun 1998]
Sliding Window ConvNet + Weighted FSM (Fixed Post-Proc)

Why Multiple Layers? The World is Compositional

- Hierarchy of representations with increasing level of abstraction
- Each stage is a kind of trainable feature transform
- Image recognition: Pixel → edge → texton → motif → part → object
- Text: Character → word → word group → clause → sentence → story
- Speech: Sample → spectral band → sound → ... → phone → phoneme → word
ConvNets are somewhat inspired by the Visual Cortex.

- The ventral (recognition) pathway in the visual cortex has multiple stages:
  - Retina - LGN - V1 - V2 - V4 - PIT - AIT ...

[picture from Simon Thorpe]

[Gallant & Van Essen]
What are ConvNets Good For

- Signals that comes to you in the form of (multidimensional) arrays.
- Signals that have strong local correlations
- Signals where features can appear anywhere
- Signals in which objects are invariant to translations and distortions.

1D ConvNets: sequential signals, text
- Text Classification
- Musical Genre Recognition
- Acoustic Modeling for Speech Recognition
- Time-Series Prediction

2D ConvNets: images, time-frequency representations (speech and audio)
- Object detection, localization, recognition

3D ConvNets: video, volumetric images, tomography images
- Video recognition / understanding
- Biomedical image analysis
- Hyperspectral image analysis
Demo of DCNN with TensorFlow

Build a DCNN to classify digital images on MNIST dataset
Key code

```python
def weight_variable(shape):
    initial = tf.truncated_normal(shape, stddev=0.1)
    return tf.Variable(initial)

def bias_variable(shape):
    initial = tf.constant(0.1, shape=shape)
    return tf.Variable(initial)
```

TensorFlow also gives us a lot of flexibility in convolution and pooling operations. How do we handle the boundaries? What is our stride size? In this example, we're always going to choose the vanilla version. Our convolutions uses a stride of one and are zero padded so that the output is the same size as the input. Our pooling is plain old max pooling over 2x2 blocks. To keep our code cleaner, let's also abstract those operations into functions.

```python
def conv2d(x, W):
    return tf.nn.conv2d(x, W, strides=[1, 1, 1, 1], padding='SAME')

def max_pool_2x2(x):
    return tf.nn.max_pool(x, ksize=[1, 2, 2, 1],
                          strides=[1, 2, 2, 1], padding='SAME')
```
First Convolutional Layer

We can now implement our first layer. It will consist of convolution, followed by max pooling. The convolution will compute 32 features for each 5x5 patch. Its weight tensor will have a shape of \([5, 5, 1, 32]\). The first two dimensions are the patch size, the next is the number of input channels, and the last is the number of output channels. We will also have a bias vector with a component for each output channel.

\[
\begin{align*}
W_{\text{conv1}} &= \text{weight\_variable}([5, 5, 1, 32]) \\
b_{\text{conv1}} &= \text{bias\_variable}([32])
\end{align*}
\]

To apply the layer, we first reshape \(x\) to a 4d tensor, with the second and third dimensions corresponding to image width and height, and the final dimension corresponding to the number of color channels.

\[
x_{\text{image}} = \text{tf.\_reshape}(x, [-1, 28, 28, 1])
\]

We then convolve \(x_{\text{image}}\) with the weight tensor, add the bias, apply the ReLU function, and finally max pool. The \(\text{max\_pool\_2x2}\) method will reduce the image size to 14x14.

\[
\begin{align*}
h_{\text{conv1}} &= \text{tf.\_nn.\_relu}(\text{conv2d}(x_{\text{image}}, W_{\text{conv1}}) + b_{\text{conv1}}) \\
h_{\text{pool1}} &= \text{max\_pool\_2x2}(h_{\text{conv1}})
\end{align*}
\]
Second Convolutional Layer

In order to build a deep network, we stack several layers of this type. The second layer will have 64 features for each 5x5 patch.

```python
W_conv2 = weight_variable([5, 5, 32, 64])
b_conv2 = bias_variable([64])

h_conv2 = tf.nn.relu(conv2d(h_pool1, W_conv2) + b_conv2)
h_pool2 = max_pool_2x2(h_conv2)
```

Densely Connected Layer

Now that the image size has been reduced to 7x7, we add a fully-connected layer with 1024 neurons to allow processing on the entire image. We reshape the tensor from the pooling layer into a batch of vectors, multiply by a weight matrix, add a bias, and apply a ReLU.

```python
W_fc1 = weight_variable([7 * 7 * 64, 1024])
b_fc1 = bias_variable([1024])

h_pool2_flat = tf.reshape(h_pool2, [-1, 7*7*64])
h_fc1 = tf.nn.relu(tf.matmul(h_pool2_flat, W_fc1) + b_fc1)
```
Dropout

To reduce overfitting, we will apply dropout before the readout layer. We create a `placeholder` for the probability that a neuron's output is kept during dropout. This allows us to turn dropout on during training, and turn it off during testing. TensorFlow's `tf.nn.dropout` op automatically handles scaling neuron outputs in addition to masking them, so dropout just works without any additional scaling.1

```python
keep_prob = tf.placeholder(tf.float32)
h_fc1_drop = tf.nn.dropout(h_fc1, keep_prob)
```

Readout Layer

Finally, we add a layer, just like for the one layer softmax regression above.

```python
W_fc2 = weight_variable([1024, 10])
b_fc2 = bias_variable([10])
y_conv = tf.matmul(h_fc1_drop, W_fc2) + b_fc2
```
Train and Evaluate

cross_entropy = tf.reduce_mean(
    tf.nn.softmax_cross_entropy_with_logits(labels=y_, logits=y_conv))
train_step = tf.train.AdamOptimizer(1e-4).minimize(cross_entropy)
correct_prediction = tf.equal(tf.argmax(y_conv, 1), tf.argmax(y_, 1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    for i in range(20000):
        batch = mnist.train.next_batch(50)
        if i % 100 == 0:
            train_accuracy = accuracy.eval(feed_dict={
                x: batch[0], y_: batch[1], keep_prob: 1.0})
            print('step %d, training accuracy %g' % (i, train_accuracy))
            train_step.run(feed_dict={x: batch[0], y_: batch[1], keep_prob: 0.5})

    print('test accuracy %g' % accuracy.eval(feed_dict={
        x: mnist.test.images, y_: mnist.test.labels, keep_prob: 1.0}))

The final test set accuracy after running this code should be approximately 99.2%.
Deep Recurrent Neural Network
Deep Autoencoder
Deep Belief Network