Lab: CudaVision – Learning Vision Systems on Graphics Cards (MA-INF 4308)

Assignment 3

13.6.2016

Prof. Sven Behnke
Dr. Seongyong Koo

universitätbonn ais

Multi-Layer Perceptron with MNIST dataset

- **Goal: How to build a general multi-layer perceptron (MLP) with TensorFlow**

- **Let's learn various techniques involved in MLP to improve performance**
  - ReLu v.s Softmax, Regularization, Initialization techniques

- **Your task is,**
  - Build your MLP model with techniques supported in TensorFlow `tf.nn.x()`
  - Find the best performance of MNIST classifier among your MLP models.

- **Theoritical reference**
  - http://www.deeplearningbook.org/contents/mlp.html

Refresher: Multi-Layer Perceptron

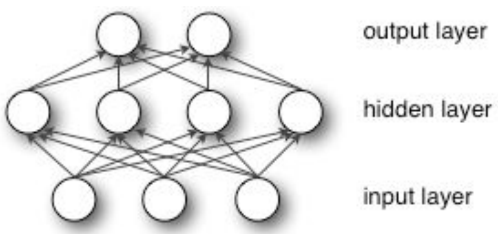
- An MLP can be viewed as a logistic regression classifier where the input is first transformed using a learnt non-linear transformation.

$$\mathbf{h} = \sigma(\mathbf{W}\mathbf{x} + \mathbf{b}) = \frac{1}{1+\exp(-\mathbf{W}\mathbf{x}-\mathbf{b})}$$

- This transformation projects the input data into a space (hidden layer) where it becomes linearly separable.

- Output layer (y) is interpreted as a vector of probabilities of input data that belongs to each class.

$$\mathbf{y} = \frac{\exp(\mathbf{V}\mathbf{h}+\mathbf{c})}{\sum_i \exp(\mathbf{V}_i\mathbf{h}+c_i)}$$



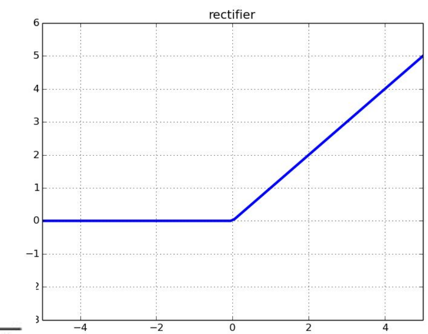
- A single hidden layer is sufficient to make MLPs a universal approximator.

- However we will see later on that there are substantial benefits to using many such hidden layers, i.e. the very premise of deep learning.

Hidden Layer Types

- Most hidden units can be described as accepting a vector of inputs, computing an affine transformation, $\boldsymbol{z} = \boldsymbol{W}^\top \boldsymbol{x} + \boldsymbol{b}$, and then applying an element-wise nonlinear function $g(\boldsymbol{z})$

- Most hidden units are distinguished from each other only by the choice of the form of the activation function.

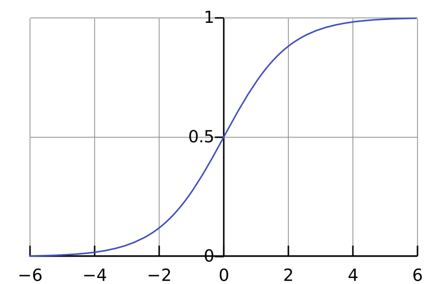


- Rectified Linear Units: $g(z) = \max\{0, z\}$

- Logistic Sigmoid $g(z) = \sigma(z) = \frac{exp(z)}{exp(z)+1} = \frac{1}{1+exp(-z)}$

- Hyperbolic Tangent $g(z) = \tanh(z)$

$$\tanh(z) = 2\sigma(2z) - 1$$



Universal Approximation Properties and Depth

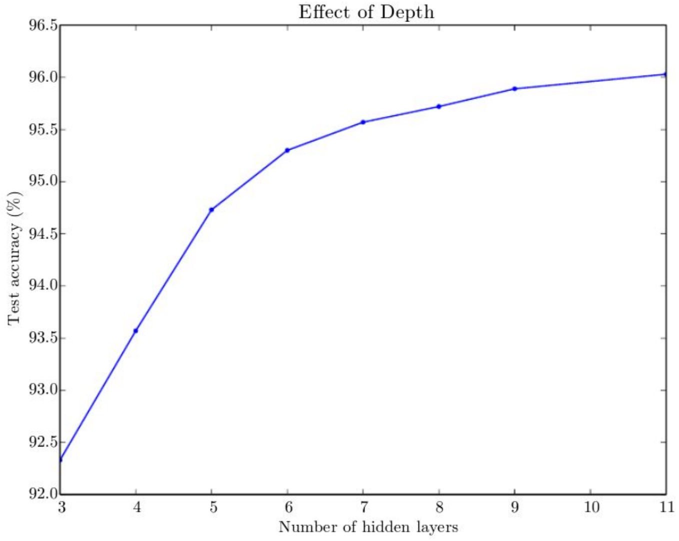
- Universal approximation theorem says that
  - there exists a network large enough with at least one hidden layer to achieve any degree of accuracy we desire, but the theorem does not say how large this network will be.

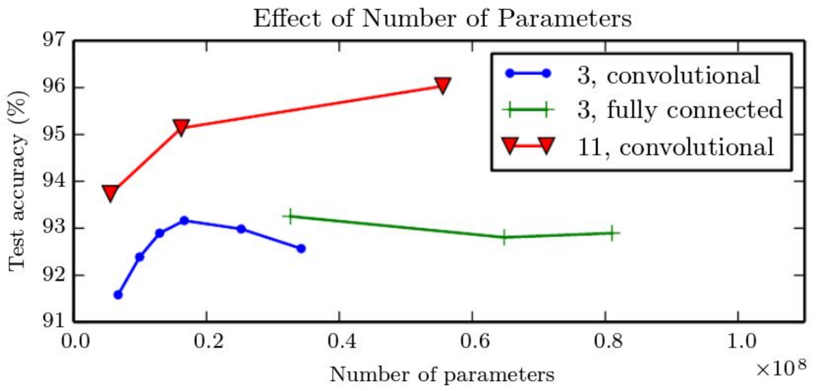
- Even if the MLP is able to represent the function, learning can fail.
  - The optimization algorithm used for training may not be able to find the value of the parameters of the desired function.
  - the training algorithm might choose the wrong function due to overfitting.

- A feedforward network with a single layer is sufficient to represent any function, but the layer may be infeasibly large and may fail to learn and generalize correctly.
  - using deeper models can reduce the number of units required to represent the desired function.

Universal Approximation Properties and Depth

- Deeper models tend to perform better. This is not merely because the model is larger.
- Empirical results showing that deeper networks generalize better when used to transcribe multi-digit numbers from photographs of addresses.
- Increasing the number of parameters in layers of convolutional networks without increasing their depth is not nearly as e ective at increasing test set performance.





universitätbonn ais

MLP implementation

- in MLPmodel.ipynb

```
sess = tf.Session()

n_input = 784 # e.g. MNIST data input (img shape: 28*28)
n_hidden = 392 # hidden layer num units (e.g. half of input units)
n_classes  = 10 # e.g. MNIST total classes (0-9 digits)

# tf Graph variables
x = tf.placeholder("float", [None, n_input], name='x')
y = tf.placeholder("float", [None, n_classes], name='y')

# Store layers weight & bias
stddev = 0.1 # <== This greatly affects accuracy!!
weights = {
    'h': tf.Variable(tf.random_normal([n_input, n_hidden], stddev=stddev)),
    'out': tf.Variable(tf.random_normal([n_hidden, n_classes], stddev=stddev))
}
biases = {
    'b': tf.Variable(tf.random_normal([n_hidden])),
    'out': tf.Variable(tf.random_normal([n_classes]))
}

# Create model
hidden_layer = tf.nn.sigmoid(tf.add(tf.matmul(x, weights['h']), biases['b']))
pred = tf.sigmoid(tf.matmul(hidden_layer, weights['out']) + biases['out'])

# Define loss
cost = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(pred, y)) # Softmax loss

# Initializing the variables
init = tf.initialize_all_variables()
sess.run(init)
```

Assignment 3

- **Finding your best MLP model for MNIST classification**
  - Train the given simple one hidden layer MLP with MNIST dataset
    - Load MNIST
    - Construct a optimizer using `tf.train.AdamOptimizer`
    - Train the model by changing learning_rate, training_epochs, batch_size.
    - What is the test accuracy of the given one-layer MLP ? (I got 0.98)
  - Build and train deeper MLPs with other activation functions, regularization, and optimization methods with
    - Increasing hidden layers
    - Changing the number of hidden units of each layer
    - ReLu activation unit by using `tf.nn.relu`
    - Dropout regularization method by using `tf.nn.dropout`
    - `tf.train.MomentumOptimizer`
  - Can you find a model performs better than the single hidden layer MLP?