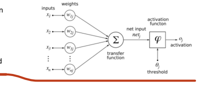
Concepts

Unit (Neurons)

A unit often refers to the activation function in a layer by which the inputs are transformed via a nonlinear activation function (for example by the logistic sigmoid function). Usually, a unit has several incoming connections and several outgoing connections.

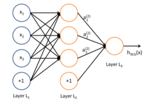


Input Layer

Comprised of multiple Real-Valued inputs. Each input must be linearly independent from each other.

Hidden Layers

Layers other than the input and output layers. A layer is the highest-level building block in deep learning. A layer is a container that usually receives weighted input, transforms it with a set of mostly non-linear functions and then passes these values as output to the next layer.



Batch Normalization

Using mini-batches of examples, as opposed to one example at a time, is helpful in several ways. First, the gradient of the loss over a mini-batch is an estimate of the gradient over the training set, whose quality improves as the batch size increases. Second, computation over a batch can be much more efficient than m computations for individual examples, due to the parallelism afforded by the modern computing platforms.

With SGD, the training proceeds in steps, and at each step we consider a mini- batch x1...m of size m. The mini-batch is used to approx-imate the gradient of the loss function with respect to the parameters.

Cost/Loss(Min) Objective(Max) Functions

Maximum Likelihood Estimation (MLE)

Many cost functions are the result of applying Maximum Likelihood. For instance, the Least Squares cost function can be obtained via Maximum Likelihood. Cross-Entropy is another example.

The likelihood of a parameter value (or vector of parameter values), θ, given outcomes x, is equal to the probability (density) assumed for those observed outcomes given those parameter values, that is

$$\mathcal{L}(\theta \mid x) = P(x \mid \theta).$$

The natural logarithm of the likelihood function, called the log-likelihood, is more convenient to work with. Because the logarithm is a monotonically increasing function, the logarithm of a function achieves its maximum value at the same points as the function itself, and hence the log-likelihood can be used in place of the likelihood in maximum likelihood estimation and related techniques.

In general, for a fixed set of data and underlying statistical model, the method of maximum likelihood selects the set of values of the model parameters that maximizes the likelihood function. Intuitively, this maximizes the "agreement" of the selected model with the observed data, and for discrete random variables it indeed maximizes the probability of the observed data under the resulting distribution. Maximum-likelihood estimation gives a unified approach to estimation, which is well-defined in the case of the normal distribution and many other problems.

$$f(x_1, x_2, \ldots, x_n \mid \theta) = f(x_1 \mid \theta) \times f(x_2 \mid \theta) \times \cdots \times f(x_n \mid \theta).$$

$$\mathcal{L}(\theta\,;x_1,\ldots,x_n) = f(x_1,x_2,\ldots,x_n \mid \theta) = \prod_{i=1}^{n} f(x_i \mid \theta).$$

$$\ln \mathcal{L}(\theta\,;x_1,\ldots,x_n) = \sum_{i=1}^{n} \ln f(x_i \mid \theta),$$

$$\hat{\ell}(\theta\,;x) = \frac{1}{n}\sum_{i=1}^{n} \ln f(x_i \mid \theta),$$

$$\{\hat{\theta}_{\mathrm{mle}}\} \subseteq \{\underset{\theta\in\Theta}{\arg\max}\ \hat{\ell}(\theta\,;x_1,\ldots,x_n)\},$$

Cross-Entropy

Cross entropy can be used to define the loss function in machine learning and optimization. The true probability pi is the true label, and the given distribution qi is the predicted value of the current model.

$$H(p,q) = \mathrm{E}_p[l_i] = \mathrm{E}_p\left[\log \frac{1}{q(x_i)}\right]$$

$$H(p,q) = \sum_{x_i} p(x_i)\, \log \frac{1}{q(x_i)}$$

$$H(p,q) = -\sum_x p(x)\, \log q(x).$$

$$L(\mathbf{w}) = \frac{1}{N}\sum_{n=1}^{N} H(p_n,q_n) = -\frac{1}{N}\sum_{n=1}^{N}\left[y_n \log \hat{y}_n + (1-y_n)\log(1-\hat{y}_n)\right]$$

Cross-entropy error function and logistic regression

Logistic

The logistic loss function is defined as:

$$V(f(\vec{x}), y) = \frac{1}{\ln 2} \ln(1 + e^{-y f(\vec{x})})$$

Quadratic

The use of a quadratic loss function is common, for example when using least squares techniques. It is often more mathematically tractable than other loss functions because of the properties of variances, as well as being symmetric: an error above the target causes the same loss as the same magnitude of error below the target. If the target is t, then a quadratic loss function is:

$$\lambda(x) = C(t-x)^2$$

0-1 Loss

In statistics and decision theory, a frequently used loss function is the 0-1 loss function

$$L(\hat{y}, y) = I(\hat{y} \neq y),$$

Hinge Loss

The hinge loss is a loss function used for training classifiers. For an intended output t = ±1 and a classifier score y, the hinge loss of the prediction y is defined as:

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

Exponential

$$E_x(\mathbf{y}, \mathbf{t}) = \tau \exp\left(\frac{1}{\tau}\sum_k (y_k - t_k)^2\right)$$

Hellinger Distance

It is used to quantify the similarity between two probability distributions. It is a type of f-divergence.

To define the Hellinger distance in terms of measure theory, let P and Q denote two probability measures that are absolutely continuous with respect to a third probability measure λ. The square of the Hellinger distance between P and Q is defined as the quantity

$$H^2(P,Q) = \frac{1}{2}\int \left(\sqrt{\frac{dP}{d\lambda}} - \sqrt{\frac{dQ}{d\lambda}}\right)^2 d\lambda.$$

Kullback-Leibler Divengence

Is a measure of how one probability distribution diverges from a second expected probability distribution. Applications include characterizing the relative (Shannon) entropy in information systems, randomness in continuous time-series, and information gain when comparing statistical models of inference.

$$D_{\mathrm{KL}}(P\|Q) = \sum_i P(i)\, \log \frac{P(i)}{Q(i)}.$$

Discrete

$$D_{\mathrm{KL}}(P\|Q) = \int_X p\, \log \frac{p}{q}\, d\mu.$$

Continuous

Itakura–Saito distance

is a measure of the difference between an original spectrum P(ω) and an approximation P^(ω) of that spectrum. Although it is not a perceptual measure, it is intended to reflect perceptual (dis)similarity.

$$D_{IS}(P(\omega), \hat{P}(\omega)) = \frac{1}{2\pi}\int_{-\pi}^{\pi}\left[\frac{P(\omega)}{\hat{P}(\omega)} - \log\frac{P(\omega)}{\hat{P}(\omega)} - 1\right] d\omega$$

https://stats.stackexchange.com/questions/154879/a-list-of-cost-functions-used-in-neural-networks-alongside-applications

https://en.wikipedia.org/wiki/Loss_functions_for_classification

Regularization

L1 norm — Manhattan Distance

L1-norm is also known as least absolute deviations (LAD), least absolute errors (LAE). It is basically minimizing the sum of the absolute differences (S) between the target value and the estimated values.

$$S = \sum_{i=1}^{n} |y_i - f(x_i)|.$$

L2 norm — Euclidean Distance

L2-norm is also known as least squares. It is basically minimizing the sum of the square of the differences (S) between the target value and the estimated values:

$$S = \sum_{i=1}^{n} (y_i - f(x_i))^2$$

Early Stopping

Early stopping rules provide guidance as to how many iterations can be run before the learner begins to over-fit, and stop the algorithm then.

Dropout

Is a regularization technique for reducing overfitting in neural networks by preventing complex co-adaptations on training data. It is a very efficient way of performing model averaging with neural networks. The term "dropout" refers to dropping out units (both hidden and visible) in a neural network

Sparse regularizer on columns

This regularizer defines an L2 norm on each column and an L1 norm over all columns. It can be solved by proximal methods.

$$R(w) = \sum_{i=1}^{D} \|W\|_{2,1}$$

Nuclear norm regularization

$R(w) = \|\sigma(W)\|_1$ where $\sigma(W)$ is the eigenvalues in the singular value decomposition of W.

Mean-constrained regularization

This regularizer constrains the functions learned for each task to be similar to the overall average of the functions across all tasks. This is useful for expressing prior information that each task is expected to share similarities with each other task. An example is predicting blood iron levels measured at different times of the day, where each task represents a different person.

$$R(f_1 \cdots f_T) = \sum_{t=1}^{T} \|f_t - \frac{1}{T}\sum_{s=1}^{T} f_s\|_{H_k}^2$$

Clustered mean-constrained regularization

This regularizer is similar to the mean-constrained regularizer, but instead enforces similarity between tasks within the same cluster. This can capture more complex prior information. This technique has been used to predict Netflix recommendations.

$R(f_1 \cdots f_T) = \sum_{r=1}^{C} \sum_{t \in I(r)} \|f_t - \frac{1}{I(r)} \sum_{s \in I(r)} f_s\|_{H_k}^2$ where $I(r)$ is a cluster of tasks.

Graph-based similarity

More general than above, similarity between tasks can be defined by a function. The regularizer encourages the model to learn similar functions for similar tasks.

$R(f_1 \cdots f_T) = \sum_{t,s=1,t\neq s}^{T} \|f_t - f_s\|^2 M_{ts}$ for a given symmetric similarity matrix M.

Activation Functions

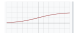
Defines the output of that node given an input or set of inputs.

Types

ReLU

$$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$$

Sigmoid / Logistic

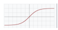


$$f(x) = \frac{1}{1+e^{-x}}$$

Binary

$$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$$

Tanh



$$f(x) = \tanh(x) = \frac{2}{1+e^{-2x}} - 1$$

Softplus

$$f(x) = \ln(1+e^x)$$

Softmax

$$f_i(\vec{x}) = \frac{e^{x_i}}{\sum_{j=1}^{J} e^{x_j}} \quad \text{for } i = 1, \ldots, J$$

Maxout

$$f(\vec{x}) = \max_i x_i$$

Leaky ReLU, PReLU, RReLU, ELU, SELU, and others.

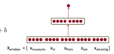
Backpropagation

Is a method used in artificial neural networks to calculate the error contribution of each neuron after a batch of data. It calculates the gradient of the loss function. It is commonly used in the gradient descent optimization algorithm. It is also called backward propagation of errors, because the error is calculated at the output and distributed back through the network layers.

$s = U^T f(Wx+b) \qquad x \in \mathbb{R}^{20\times 1}, W \in \mathbb{R}^{8\times 20}, U \in \mathbb{R}^{8\times 1}$

$$s = U^T a, \quad a = f(z), \quad z = Wx + b$$

$$J = \max(0, 1 - s + s_c) \qquad s = U^T f(Wx+b), \quad s_c = U^T f(Wx_c+b)$$



Neural Network taking 4 dimension vector representation of words.

In this method, we reuse partial derivatives computed for higher layers in lower layers, for efficiency.

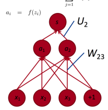
$$\frac{\partial J}{\partial U} = 1\{1 - s + s_c > 0\}\,(-f(Wx+b) + f(Wx_c+b))$$

$$\frac{\partial J}{\partial s} = 1\{1 - s + s_c > 0\}\,(-a + a_c)$$

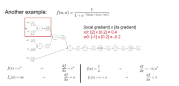
$$\frac{\partial s}{\partial x} = W^T \delta$$

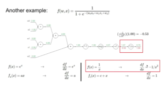
$$\begin{aligned}\frac{\partial s}{\partial x_j} &= \sum_{i=1}^{2} \frac{\partial s}{\partial a_i}\frac{\partial a_i}{\partial x_j} \\ &= \sum_{i=1}^{2} \frac{\partial U^T a}{\partial a_i}\frac{\partial a_i}{\partial x_j} \\ &= \sum_{i=1}^{2} U_i \frac{\partial f(W_{i\cdot}x+b)}{\partial x_j} \\ &= \sum_{i=1}^{2} \underbrace{U_i f'(W_{i\cdot}x+b)}\frac{\partial W_{i\cdot}x}{\partial x_j} \\ &= \sum_{i=1}^{2} \delta_i W_{ij} \\ &= \delta^T W_{\cdot j}\end{aligned}$$

$$z_i = W_{i\cdot}x + b_i = \sum_{j=1}^{3} W_{ij}x_j + b_i, \qquad a_i = f(z_i)$$

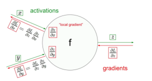


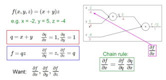
Intuition for backpropagation



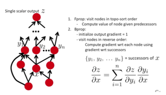


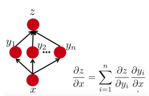
Another Example (Circuits)

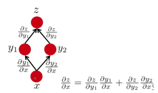


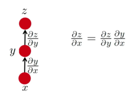


Simple Example (Circuits)









Simple Example (Flowgraphs)

Learning Rate

Neural networks are often trained by gradient descent on the weights. This means at each iteration we use backpropagation to calculate the derivative of the loss function with respect to each weight and subtract it from that weight.

However, if you actually try that, the weights will change far too much each iteration, which will make them "overcorrect" and the loss will actually increase/diverge. So in practice, people usually multiply each derivative by a small value called the "learning rate" before they subtract it from its corresponding weight.

Tricks

Simplest recipe: keep it fixed and use the same for all parameters.

$$\theta^{new} = \theta^{old} - \alpha \nabla_\theta J_t(\theta)$$

Better results by allowing learning rates to decrease Options:

- Reduce by 0.5 when validation error stops improving
- Reduction by O(1/t) because of theoretical convergence guarantees, with hyper-parameters ε_0 and τ and t is iteration numbers.

$$\alpha = \frac{\alpha_0 \tau}{\max(t, \tau)}$$

- Better yet: No hand-set learning of rates by using AdaGrad

Optimization

Gradient Descent

Is a first-order iterative optimization algorithm for finding the minimum of a function. To find a local minimum of a function using gradient descent, one takes steps proportional to the negative of the gradient (or of the approximate gradient) of the function at the current point. If instead one takes steps proportional to the positive of the gradient, one approaches a local maximum of that function; the procedure is then known as gradient ascent.

Stochastic Gradient Descent (SGD)

Gradient descent uses total gradient over all examples per update, SGD updates after only 1 or few examples:

$$\theta^{new} = \theta^{old} - \alpha \nabla_\theta J_t(\theta)$$

Mini-batch Stochastic Gradient Descent (SGD)

Gradient descent uses total gradient over all examples per update, SGD updates after only 1 example

Momentum

Idea: Add a fraction v of previous update to current one. When the gradient keeps pointing in the same direction, this will increase the size of the steps taken towards the minimum.

Adagrad

Adaptive learning rates for each parameter

Weight Initialization

All Zero Initialization

In the ideal situation, with proper data normalization it is reasonable to assume that approximately half of the weights will be positive and half of them will be negative. A reasonable-sounding idea then might be to set all the initial weights to zero, which you expect to be the "best guess" in expectation.

But, this turns out to be a mistake, because if every neuron in the network computes the same output, then they will also all compute the same gradients during back-propagation and undergo the exact same parameter updates. In other words, there is no source of asymmetry between neurons if their weights are initialized to be the same.

Initialization with Small Random Numbers

Thus, you still want the weights to be very close to zero, but not identically zero. In this way, you can random these neurons to small numbers which are very close to zero, and it is treated as symmetry breaking. The idea is that the neurons are all random and unique in the beginning, so they will compute distinct updates and integrate themselves as diverse parts of the full network.

The implementation for weights might simply drawing values from a normal distribution with zero mean, and unit standard deviation. It is also possible to use small numbers drawn from a uniform distribution, but this seems to have relatively little impact on the final performance in practice.

Calibrating the Variances

One problem with the above suggestion is that the distribution of the outputs from a randomly initialized neuron has a variance that grows with the number of inputs. It turns out that you can normalize the variance of each neuron's output to 1 by scaling its weight vector by the square root of its fan-in (i.e., its number of inputs)

This ensures that all neurons in the network initially have approximately the same output distribution and empirically improves the rate of convergence. The detailed derivations can be found from Page. 18 to 23 of the slides. Please note that, in the derivations, it does not consider the influence of ReLU neurons.

Architectures

Feed Forward

Is an artificial neural network wherein connections between the units do not form a cycle. In this network, the information moves in only one direction, forward, from the input nodes, through the hidden nodes (if any) and to the output nodes. There are no cycles or loops in the network.

Kinds

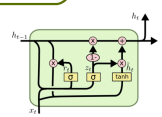
Single-Layer Perceptron

The inputs are fed directly to the outputs via a series of weights. By adding an Logistic activation function to the outputs, the model is identical to a classical Logistic Regression model.

Multi-Layer Perceptron

This class of networks consists of multiple layers of computational units, usually interconnected in a feed-forward way. Each neuron in one layer has directed connections to the neurons of the subsequent layer. In many applications the units of these networks apply a sigmoid function as an activation function.

LSTMs

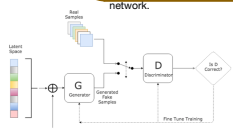


$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$
$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$
$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$
$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

An LSTM is well-suited to learn from experience to classify, process and predict time series given time lags of unknown size and bound between important events. Relative insensitivity to gap length gives an advantage to LSTM over alternative RNNs, hidden Markov models and other sequence learning methods in numerous applications.

Long short-term memory - It is a type of recurrent (RNN), allowing data to flow both forwards and backwards within the network.

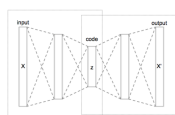
GANs



GANs or Generative Adversarial Networks are a class of artificial intelligence algorithms used in unsupervised machine learning, implemented by a system of two neural networks contesting with each other in a zero-sum game framework.

Auto-Encoders

The aim of an autoencoder is to learn a representation (encoding) for a set of data, typically for the purpose of dimensionality reduction. Recently, the autoencoder concept has become more widely used for learning generative models of data.



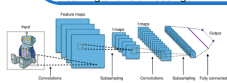
Is an artificial neural network used for unsupervised learning of efficient codings.

Convolutional Neural Networks (CNN)

Pooling

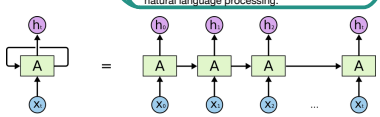
Convolution

Subsampling



They have applications in image and video recognition, recommender systems and natural language processing.

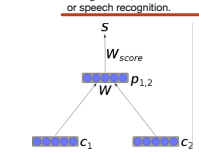
RNNs (Recurrent)



Is a class of artificial neural network where connections between units form a directed cycle. This allows it to exhibit dynamic temporal behavior. Unlike feedforward neural networks, RNNs can use their internal memory to process arbitrary sequences of inputs.

This makes them applicable to tasks such as unsegmented, connected handwriting recognition or speech recognition.

RNNs (Recursive)



RNNs have been successful for instance in learning sequence and tree structures in natural language processing, mainly phrase and sentence continuous representations based on word embedding.

Is a kind of deep neural network created by applying the same set of weights recursively over a structure, to produce a structured prediction over variable-size input structures, or a scalar prediction on it, by traversing a given structure in topological order.

Strategy

1.Select Network Structure appropriate for problem

Structure: Single words, fixed windows, sentence based, document level; bag of words, recursive vs. recurrent, CNN

Nonlinearity (Activation Functions)

2.Check for implementation bugs with gradient checks

1.Implement your gradient

2.Implement a finite difference computation by looping through the parameters of your network, adding and subtracting a small epsilon (~10-4) and estimate derivatives

$$f'(\theta) \approx \frac{J\left(\theta^{(i+)}\right) - J\left(\theta^{(i-)}\right)}{2\epsilon} \qquad \theta^{(i+)} = \theta + \epsilon \times e_i$$

3.Compare the two and make sure they are almost the same

Using Gradient Checks

If you gradient fails and you don't know why?

Simplify your model until you have no bug!

What now? Create a very tiny synthetic model and dataset

Example: Start from simplest model then go to what you want:

Only softmax on fixed input

Backprop into word vectors and softmax Add

single unit single hidden layer

Add multi unit single layer

Add second layer single unit, add multiple units, bias • Add one softmax on top, then two softmax layers

Add bias

3.Parameter initialization

Initialize hidden layer biases to 0 and output (or reconstruction) biases to optimal value if weights were 0 (e.g., mean target or inverse sigmoid of mean target).

Initialize weights ~ Uniform(–r, r), r inversely proportional to fan-in (previous layer size) and fan-out (next layer size):

$$\sqrt{6/(\text{fan-in} + \text{fan-out})}$$

4.Optimization

Gradient Descent

Is a first-order iterative optimization algorithm for finding the minimum of a function. To find a local minimum of a function using gradient descent, one takes steps proportional to the negative of the gradient (or of the approximate gradient) of the function at the current point. If instead one takes steps proportional to the positive of the gradient, one approaches a local maximum of that function; the procedure is then known as gradient ascent.

Stochastic Gradient Descent (SGD)

Gradient descent uses total gradient over all examples per update, SGD updates after only 1 or few examples:

$$\theta^{new} = \theta^{old} - \alpha \nabla_\theta J_t(\theta)$$

Ordinary gradient descent as a batch method is very slow, should never be used. Use 2nd order batch method such as L-BFGS.

On large datasets, SGD usually wins over all batch methods. On smaller datasets L-BFGS or Conjugate Gradients win. Large-batch L-BFGS extends the reach of L-BFGS [Le et al. ICML 2011].

Mini-batch Stochastic Gradient Descent (SGD)

Gradient descent uses total gradient over all examples per update, SGD updates after only 1 example

Most commonly used now, Size of each mini batch B: 20 to 1000

$$\theta^{new} = \theta^{old} - \alpha \nabla_\theta J_{t:t+B}(\theta)$$

Helps parallelizing any model by computing gradients for multiple elements of the batch in parallel

Momentum

Idea: Add a fraction v of previous update to current one. When the gradient keeps pointing in the same direction, this will increase the size of the steps taken towards the minimum.

Reduce global learning rate when using a lot of momentum

Update Rule

$$v = \mu v - \alpha \nabla_\theta J_t(\theta)$$
$$\theta^{new} = \theta^{old} + v$$

v is initialized at 0

Momentum often increased after some epochs (0.5 à 0.99)

Adagrad

Adaptive learning rates for each parameter!

Learning rate is adapting differently for each parameter and rare parameters get larger updates than frequently occurring parameters. Word vectors!

Let $g_{t,i} = \frac{\partial}{\partial \theta_i^t} J_t(\theta)$, then: $\theta_{t,i} = \theta_{t-1,i} - \frac{\alpha}{\sqrt{\sum_{\tau=1}^{t} g_{\tau,i}^2}} g_{t,i}$

5.Check if the model is powerful enough to overfit

If not, change model structure or make model "larger"

If you can overfit: Regularize to prevent overfitting:

Simple first step: Reduce model size by lowering number of units and layers and other parameters

Standard L1 or L2 regularization on weights

Early Stopping: Use parameters that gave best validation error

Sparsity constraints on hidden activations, e.g., add to cost:

Dropout

Training time: at each instance of evaluation (in online SGD-training), randomly set 50% of the inputs to each neuron to 0

Test time: halve the model weights (now twice as many) This prevents feature co-adaptation: A feature cannot only be useful in the presence of particular other features

In a single layer: A kind of middle-ground between Naïve Bayes (where all feature weights are set independently) and logistic regression models (where weights are set in the context of all others)

Can be thought of as a form of model bagging

It also acts as a strong regularizer

TensorFlow

Intuition

TensorFlow is a deep learning library recently open-sourced by Google. It provides primitives for defining functions on tensors and automatically computing their derivatives, expressed as a graph.

The Tensorflow Graph is build to contain all placeholders for X and y, all variables for W's and b's, all mathematical operations, the cost function, and the optimisation procedure. Then, at runtime, the values for the data are fed into that Graph, by placing the data batches in the placeholders and running the Graph.

Each node in the Graph can then be connected to each other node over the network, and thus running Tensorflow models can be parallelised.

Tensorboard

TensorFlow has some neat built-in visualization tools (TensorBoard).

Phases

1. Construction

Assembles a computational graph

The computation graph has no numerical value until evaluated.

All computations add nodes to global default graph

2. Execution

A Session object encapsulates the environment in which Tensor objects are evaluated

Uses a session to execute ops in the graph

Declared variables must be initialised before they have values.

When you train a model you use variables to hold and update parameters. Variables are in-memory buffers containing tensors.

Main Components

Variables

Stateful nodes that output their current value, their state is retained across multiple executions of the graph.

Mostly Parameters we're interested in tuning, such as Weights (W) and Biases (b).

Sharing

Variables can be shared by Explicitly passing tf.Variable objects around, or...

Implicitly wrapping tf.Variable objects within tf.variable_scope objects.

Scopes

tf.variable_scope()

```
with tf.variable_scope("foo"):
    v = tf.get_variable("v", shape=[1])  # v.name == "foo/v:0"

with tf.variable_scope("foo", reuse=True):
    v1 = tf.get_variable("v")  # Shared variable found!

with tf.variable_scope("foo", reuse=False):
    v1 = tf.get_variable("v")  # CRASH foo/v:0 already exists!
```

Provides simple name spacing to avoid cases when querying

tf.get_variable()

Creates/Access variables from a variable scope

Placeholders

Nodes whose value is fed at execution time.

Inputs, Features (X) and Labels (y)

Mathematical Operations

MatMul, Add, ReLU, etc.

Graph

They are Operations, containing any number of inputs and outputs.

Nodes

The tensors that flow between the nodes. Edges It a

Session

binding to a particular execution context: CPU, GPU.

Running a Session

Inputs

Fetches

List of graph nodes. Returns the output of these nodes.

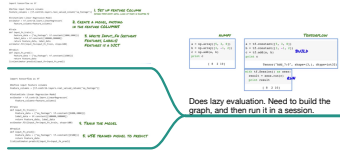
Feeds

Dictionary mapping from graph nodes to concrete values.

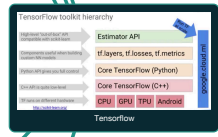
Specified the value of each graph node given in the dictionary.

Comparison to Numpy

Does lazy evaluation. Need to build the graph, and then run it in a session.



Packages



tf

Main Steps

1. 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
```

2. Define Target

```
y_ = tf.placeholder(tf.float32, [None, 10])
```

3. Define Loss function and Optimizer

```
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)
```

4. Define the Session and Initialise Variables

```
sess = tf.InteractiveSession()
tf.global_variables_initializer().run()
```

5. Train the Model

```
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})
```

6. 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}))
```

tf.estimator

TensorFlow's high-level machine learning API (tf.estimator) makes it easy to configure, train, and evaluate a variety of machine learning models.

- tf.estimator.LinearClassifier: Constructs a linear classification model.
- tf.estimator.LinearRegressor: Constructs a linear regression model.
- tf.estimator.DNNClassifier: Construct a neural network classification model.
- tf.estimator.DNNRegressor: Construct a neural network regression model.
- tf.estimator.DNNLinearCombinedClassifier: Construct a neural network and linear combined classification model.
- tf.estimator.DNNRegressor: Construct a neural network and linear combined regression model.

```
estimator = tf.estimator.inputs.pandas_input_fn(x=pd.DataFrame({'k': np.ones(1)}), y=pd.Series(np.ones(1)), num_epochs=1)
classifier = tf.estimator.LinearClassifier(feature_columns=[tf.feature_column.numeric_column('k')])
classifier.train(input_fn=lambda: _input_fn(), steps=100)
scores = classifier.evaluate(input_fn=lambda: _input_fn(), steps=100)
print("Accuracy (after training): %s" % scores['accuracy'])
```

```
age = tf.contrib.layers.real_valued_column("age")
height = tf.contrib.layers.real_valued_column("height")
first_layer = tf.contrib.layers.create_feature_spec_for_parsing([age, height])
input_fn = tf.contrib.learn.io.build_parsing_serving_input_fn(first_layer)
age = tf.contrib.layers.real_valued_column("age")
height = tf.contrib.layers.real_valued_column("height")
input_fn = tf.contrib.learn.io.build_parsing_serving_input_fn(
    tf.contrib.layers.create_feature_spec_for_parsing([age, height]))
```

Main Steps

1. Define Feature Columns

FeatureColumns are the primary way of encoding features for pre-canned tf.learn Estimators.

When using FeatureColumns with tf.learn models, the type of feature column you should choose depends on the feature type and the model type.

- Categorical
  - Continuous Features
    - Can be represented by real_valued_column
  - Categorical Features
    - Can be represented by any sparse_column_with_* column (sparse_column_with_keys, sparse_column_with_vocabulary_file, sparse_column_with_hash_bucket, sparse_column_with_integerized_feature
- Numerica

2. Define your Layers, or use a prebuilt model

```
model_dir = tempfile.mkdtemp()
m = tf.contrib.learn.LinearClassifier(feature_columns=[
    workclass, occupation, native_country,
    education, marital_status, sex,
    race, age_buckets, education_occupation,
    age_race_occupation, gender_x_relationship,
    logistic_pets, logistic_texts, age_texts],
    model_dir=model_dir)
```

Using a pre-built Logistic Regression Classifier

3. Write the input_fn function

```
def input_fn(df):
  continuous_cols = {k: tf.constant(df[k].values) for k in CONTINUOUS_COLUMNS}
  categorical_cols = {k: tf.SparseTensor(
      indices=[[i, 0] for i in range(df[k].size)],
      values=df[k].values,
      dense_shape=[df[k].size, 1]) for k in CATEGORICAL_COLUMNS}
  feature_cols = dict(continuous_cols.items() + categorical_cols.items())
  label = tf.constant(df[LABEL_COLUMN].values)
  return feature_cols, label

def train_input_fn():
  return input_fn(train)

def eval_input_fn():
  return input_fn(test)
```

This function holds the actual data (features and labels). Features is a python dictionary.

4. Train the model

```
m.fit(input_fn=train_input_fn, steps=200)
```

Using the fit function, on the input_fn. Notice that the feature columns are fed to the model as arguments.

5. Predict and Evaluate

```
results = m.evaluate(input_fn=eval_input_fn, steps=1)
for key in sorted(results):
    print("%s: %s" % (key, results[key]))
```

Using the eval_input_fn defined previously.