(https://www.nvidia.com/dli)

TensorFlow로 이미지 분할하기

이미지 내의 개별 개체들을 감지하여 관심 있는 부분을 이미지 조각으로 분리해 내는 기술, 그 이상의 이미지 분석용 딥러닝 애플리케이션들이 다양하게 존재합니다. 예를 들어, 의료용 영상 분석의 경우, 특정 대상 장기의 이미지를 구별해 내어, 조직 및 혈액의 유형 및 비정상 세포에 해당하는 이미지 픽셀을 분리해 내는 것이 매우 중요할 때가 많습니다. 이번 실습에서는 의료 영상 데이터셋을 가지고 이미지 분할 네트워크(image segmentation network)를 학습시키고 평가하기 위해 TensorFlow (https://www.tensorflow.org)의 머신 러닝 프레임워크를 사용하겠습니다.

이번 실습은 Jonathan Bentz (Twitter 팔로우하기: @jnbntz (https://twitter.com/jnbntz))가 구성하였습니다.

시작하기 전에 WebSockets (http://en.wikipedia.org/wiki/WebSocket)이 시스템에서 작동하는지 확인하기 위해, 아래 구문을 여러분의 마우스로 클릭하여 포커스 한 후, Ctrl-Enter를 누르거나, 상단의 도구모음에서 재생(play) 버튼을 눌러 실행하십시오. 다 잘 진행된다면, 아래의 일부 출력이 회색 셀로 표현되는 것을 볼 수 있을 것입니다. 잘 안되는 경우에는, 자기주도학습 랩 문제해결 FAQ (https://developer.nvidia.com/self-paced-labs-faq#Troubleshooting)를 참조하여 문제를 해결하십시오.

In [1]:

```
print "The answer should be three: " + str(1+2)
```

```
The answer should be three: 3
```

아래를 실행하여 서버에서 실행 중인 GPU에 대한 정보를 표시하십시오.

In [2]:

```
!nvidia-smi
```

```
Sun Feb 16 13:31:21 2020
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 396.26                 Driver Version: 396.26                    |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  Tesla K80           Off  | 00000000:00:1E.0 Off |                    0 |
| N/A   45C    P0    70W / 149W |    135MiB / 11441MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                       GPU Memory |
|  GPU       PID   Type   Process name                             Usage      |
|=============================================================================|
+-----------------------------------------------------------------------------+
```

마지막으로 다음을 실행하여 이 실습에서 사용하는 TensorFlow의 버전을 확인할 수 있습니다.

In [3]:

```
!python -c 'import tensorflow as tf; print(tf.__version__)'
```

```
1.7.0
```

만약 여러분이 NVIDIA의 IPython Notebook 기반 자기주도 학습 랩을 사용해 본 적이 없다면, 다음의 짧은 YouTube 동영상 (http://www.youtube.com/embed/ZMrDaLSFqpY)을 시청할 것을 권장합니다.

이미지 분할

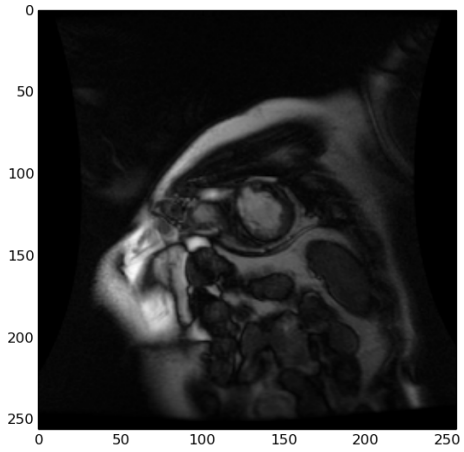
이번에는 이미지 분할(또는 시멘틱 분할- semantic segmentation이라고도 함)을 위한 일련의 실습을 수행하겠습니다. 시멘틱 분할은 각 픽셀을 특정 클래스에 배치하는 작업으로, 어떤 의미에서 이것은 전체 이미지가 아닌 픽셀 단위로 이미지를 분류하는 것입니다. 이 실습에서는 특정 픽셀이 좌심실(LV)의 일부인지 여부에 따라 심장 MRI 영상의 각 픽셀을 분류하겠습니다.

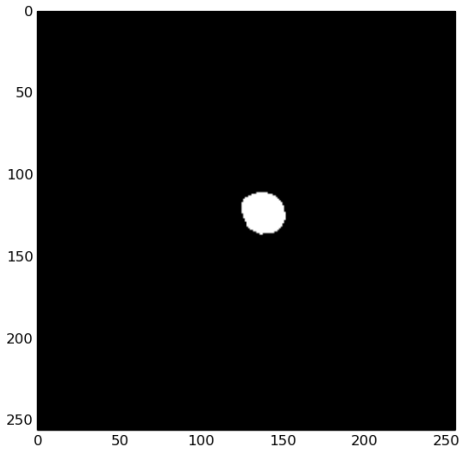
이번 랩은 딥러닝에 대한 소개나 콘볼루션 신경망(convolutional neural networks (CNN))에 대한 복잡한 수학 공식을 알려주기 위한 실습이 아닙니다. 우리는 여러분이 적어도 전진과 forward and backpropagation, activations, SGD, convolutions, pooling, bias 등과 같은 개념을 포함하는 신경망에 대한 기본적인 이해를 이미 알고 있다고 가정하겠습니다. 만약 이미 CNN를 접해본 적이 있으며, 이미지 인식 작업에 대한 이해가 있다면 매우 도움이 될 것입니다. 이번 실습은 Google의 TensorFlow 머신 러닝 프레임워크를 사용할 예정이라, 만약 여러분이 Python과 TensorFlow를 사용해본 경험이 있다면 매우 도움이 되겠지만 반드시 필요한 조건은 아닙니다. 이번 실습에서 여러분이 수행할 대부분의 작업들은 코딩 그 자체가 아니라, TensorFlow를 사용하여 학습 및 평가 작업을 수행하는 데 있습니다.

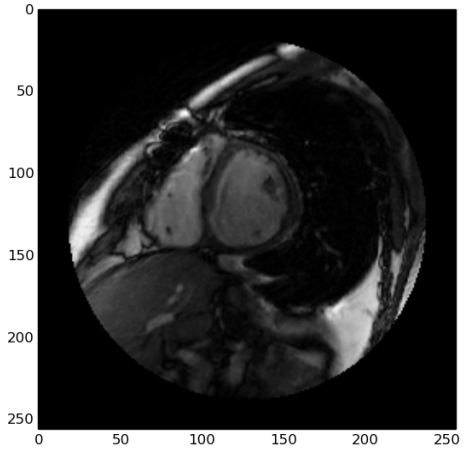
입력 데이터셋

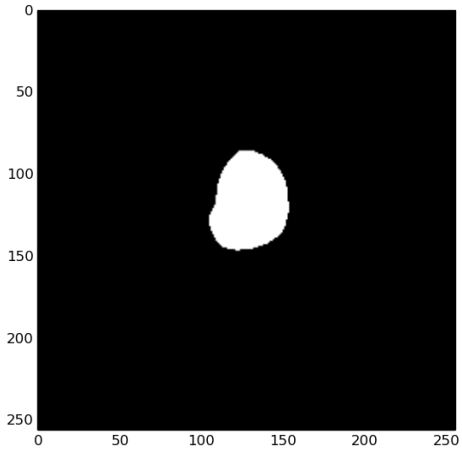
사용할 데이터셋은 전문적인 라벨이 부착된 일련의 심장 영상(특히 MRI SAX (Short-axis) 스캔 이미지)입니다. 전체 인용 정보는 참고문헌 [1, 2, 3]을 참조하십시오.

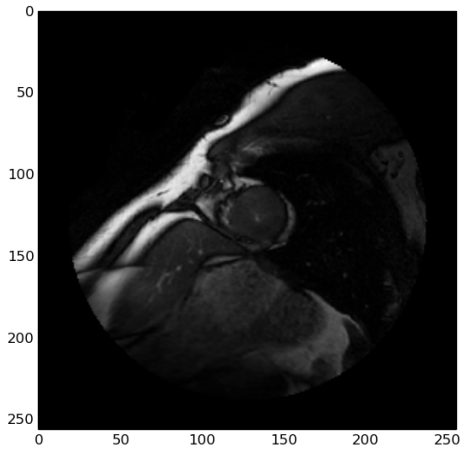
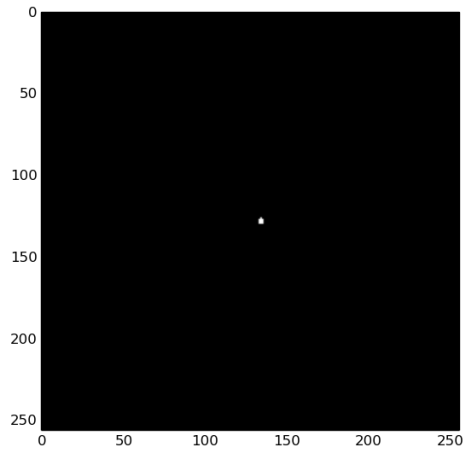
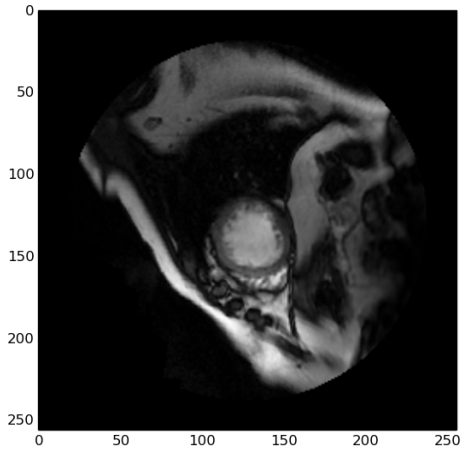
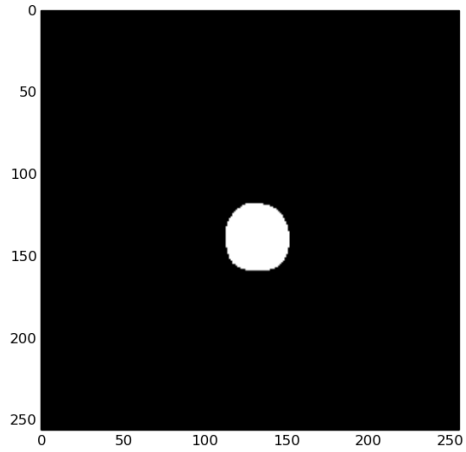
데이터의 4가지 대표적인 예는 다음과 같습니다. 각 이미지의 행은 데이터의 인스턴스(instance)입니다. 왼쪽은 MRI 영상이고 오른쪽은 전문적으로 분할된 영역(종종 윤곽선이라고 함)입니다. LV의 일부 이미지는 흰색으로 표시됩니다. LV의 크기는 이미지에 따라 다르지만 일반적으로 LV는 전체 이미지에서 비교적 작은 영역을 차지합니다.









원본 이미지에서 데이터를 추출한 다음 TensorFlow로 보내기 위해 이 이미지들을 준비하는 작업은 이번 실습에 포함되지 않습니다. 데이터 준비는 머신 러닝의 사소한 일부이기 때문에 이번 실습의 범위를 벗어나기 때문입니다.

하지만, 상세한 내용에 관심있는 분들을 위해, 이미지 추출법에 대해 이전 Kaggle 경진대회 (https://www.kaggle.com/c/second-annual-data-science-bowl/overview/deep-learning-tutorial)에서 가이드 및 일부 코드를 가져왔습니다. 우리는 이미 가져온 이미지를 TensorFlow records (TFRecords)로 변환하여 파일에 저장했습니다. TFRecords (https://www.tensorflow.org/programmers_guide/reading_data)는 TensorFlow에서 제공하는 특정 파일 형식으로, 다중 스레드(multi-threaded) 데이터 읽기 및 정교한 사전처리 (예를 들어, 학습 데이터 확장(augmenting) 및 랜덤화(randomizing) 등)와 같은 데이터 관리를 위해 TensorFlow에 내장된 기능들을 사용할 수 있게 됩니다.

이미지 자체는 원래 256 x 256 그레이스케일 DICOM (https://en.wikipedia.org/wiki/DICOM) 형식이며 의료 영상에서 일반적인 형식입니다. 라벨은 256 x 256 x 2 크기의 tensor이며, 마지막 숫자 2는 픽셀이 두 클래스 중 하나이기 때문입니다. 실습용 데이터셋은 234개의 이미지로 구성되어 있고, 검증용 데이터셋(학습에 사용되지 않지만, 모델의 정확성 확인을 위해 사용되는 데이터)은 26개 이미지입니다.

TensorFlow 기반 딥러닝

이 실습은 현재 공개적으로 사용 가능한 딥 러닝 프레임워크를 소개하기 위해 설계된 일련의 자체 학습용 랩의 일부로, TensorFlow는 Google (https://www.google.com)에서 개발한 프레임워크이며, Google의 수많은 연구자들과 제품군에서 사용되고 있습니다. TensorFlow는 머신 인텔리전스를 위한 오픈 소스 소프트웨어 라이브러리이며, 컴퓨팅 결과값은 tensor(이름 입력)에서 작동하는 데이터의 흐름 그래프로 표현됩니다. 만약 여러분도 이런 방식으로 표현할 수 있다면, 여러분의 알고리즘도 TensorFlow 프레임워크에서 실행될 수 있다는 뜻입니다.

TensorFlow는 CPU 및 GPU에서 실행될 수 있고, 워크 스테이션이나 서버, 모바일 기기에서 모델을 구현하여 사용할 수 있다는 점에서 휴대성이 좋습니다. 현재 TensorFlow는 여러분의 컴퓨팅을 Python 또는 C++로 표현하는 옵션을 제공하며, 다른 종류의 언어들 (https://www.tensorflow.org/api_docs/)도 다양하게 지원합니다. 일반적으로 TensorFlow는 Python에서 학습 및 테스트를 수행하며, 모델을 완성한 후에는 C++로 배포될 수 있습니다.

TensorFlow는 CPU와 GPU 모두에서 성능을 발휘하도록 설계 및 제작되었습니다. 단일 TensorFlow 실행 중에도 필요한 경우 CPU 및 GPU에 서로 다른 작업을 명시적으로 할당할 수 있어 높은 유연성을 갖습니다. GPU에서 실행 시, TensorFlow는 cuDNN (https://developer.nvidia.com/cudnn)을 비롯한 여러 GPU 라이브러리를 활용하여 최신 GPU에서 사용가능한 가장 높은 성능을 끌어낼 수 있습니다.

이 실습의 의의 중 하나는 TensorFlow에 대한 친숙도를 높이는 것입니다. 이 짧은 실습을 통해 TensorFlow의 모든 기능과 옵션에 대해 설명할 수는 없지만, 이번 실습을 완료한 후에는 여러분의 머신 러닝 관련 문제들을 해결하기 위해 TensorFlow를 편안하게 느끼고, 좋은 아이디어를 가지고, TensorFlow를 잘 활용할 수 있게 되기를 바랍니다. TensorFlow에 대한 전반적인 이해는 TensorFlow 웹 사이트 (https://www.tensorflow.org)와 백서 (http://download.tensorflow.org/paper/whitepaper2015.pdf) 및 GitHub 사이트 (https://github.com/tensorflow/tensorflow)를 활용해 주시기 바랍니다.

TensorFlow의 기본 알기

TensorFlow는 여러분의 기호에 따라 다양하게 사용될 수 있습니다. 학습과 관련한 작업을 설계할 때 가장 일반적인 방법은 TensorFlow Python API를 사용하는 것입니다. TensorFlow에서 머신 러닝 관련 작업을 실행하려면 (최소한) 두 단계가 필요합니다.

데이터 흐름 그래프 (Data Flow Graph)

먼저 여러분이 수행하려는 연산들이 정확히 무엇인지에 대한 조건과 순서에 따라 데이터 흐름 그래프를 작성합니다. TensorFlow API를 사용하여 convolutions, activations, pooling 등과 같이 TensorFlow의 기본 작업들을 활용하거나, 여러분은 레이어 별로 신경망을 구축합니다. 전체 프로세스 중 이 단계에서는 어떠한 실제 데이터 계산이 이루어지지 않으며, 여러분이 지정한 그래프만 구성하게 됩니다. 그래프를 작성할 때에는 (TensorFlow lexicon)의 각 변수(`Variable`)들을 지정해야 합니다. 특정 데이터를 변수로 지정하는 것은 TensorFlow에게 이것은 "학습"할 파라미터, 즉 학습이 진행됨에 따라 업데이트 되어야 할 가중치라는 것을 알려주는 것입니다.

세션 (Session)

신경망을 데이터 흐름 그래프로 정의하면 세션(`Session`)이 시작됩니다. 이 메커니즘은 이전에 생성된 그래프에 대한 입력 데이터와 교육 매개 변수를 지정한 다음 계산이 진행되는 메커니즘입니다. 일반적으로 이 두 단계는 그래프를 변경할 때마다, 즉 그래프를 업데이트하고 새 세션을 시작할 때마다 반복됩니다.

샘플 워크플로우

모델을 학습시키고 평가하는 워크플로우 샘플의 예는 다음과 같습니다.

1. 입력 데이터 준비하기 - 입력 데이터는 Numpy 배열일 수 있지만 매우 큰 데이터셋의 경우 TensorFlow는 TFRecords라는 특수 형식을 제공합니다.

2. 컴퓨팅 그래프를 만들기 - 추론, 손실 및 훈련 노드와 같은 특수 노드를 포함한 신경망 그래프를 만듭니다.
3. 모델 학습시키기 - 입력 데이터를 TensorFlow 세션의 그래프에 삽입하고 데이터 입력을 반복합니다. 배치 크기, 에포크(epoch) 수, 학습 속도 등을 사용자 지정합니다.
4. 모델 평가하기 – 처음 접하는 데이터에 대한 추론을 실행해 보고, 적절한 메트릭을 기반으로 모델의 정확도를 평가합니다.

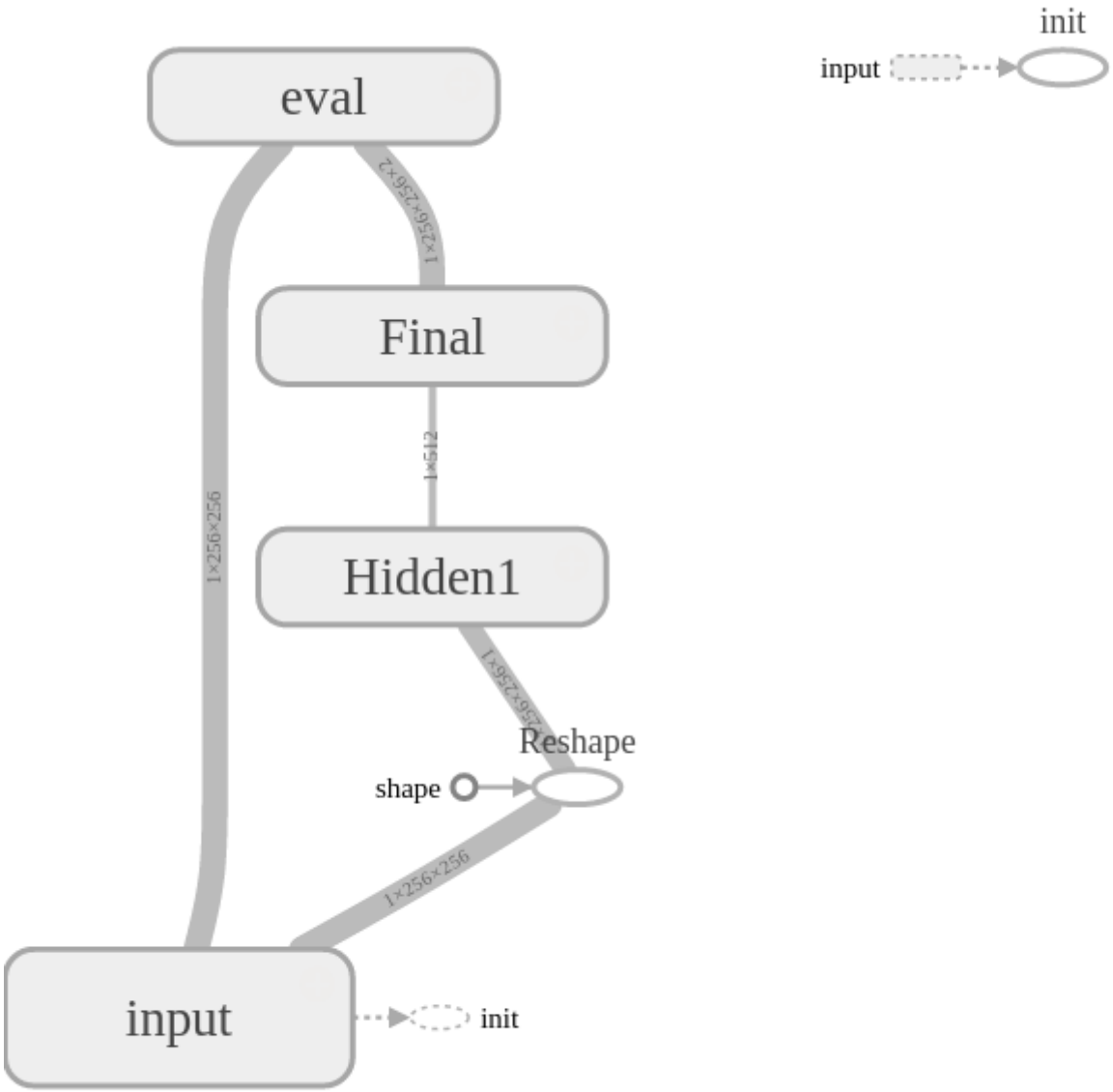
TensorBoard

TensorFlow는 프로그램의 많은 부분을 시각화 할 수 있는 TensorBoard (https://www.tensorflow.org/get_started/summaries_and_tensorboard)라는 기능이 풍부한 도구를 제공합니다. TensorBoard에서 컴퓨팅 그래프를 시각적으로 표현하고 손실, 정확도 및 학습 속도와 같은 다양한 계산 메트릭을 표시할 수 있습니다. 기본적으로 TensorFlow를 실행하는 동안 생성되는 모든 데이터는 프로그램에 몇 가지 추가 API 호출을 추가하여 TensorBoard에서 시각적으로 표시할 수 있습니다.

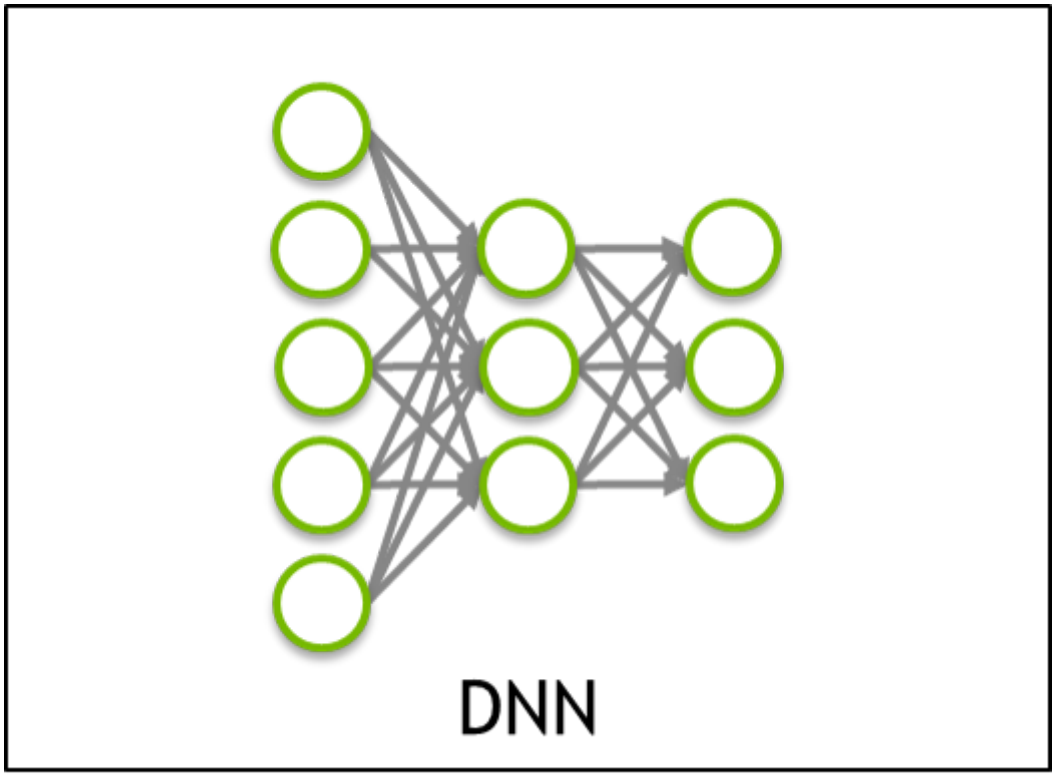
예를 들어, 한 개의 숨겨진 레이어가 있는 신경망을 생성하는 다음 코드의 일부를 봅시다. (지금은 코드의 세부 정보까지 걱정하지 마십시오).

```
    with tf.name_scope('Hidden1'):
        W_fc = tf.Variable(tf.truncated_normal( [256*256, 512],
                     stddev=0.1, dtype=tf.float32), name='W_fc')
        flatten1_op = tf.reshape( images_re, [-1, 256*256])
        h_fc1 = tf.matmul( flatten1_op, W_fc )

    with tf.name_scope('Final'):
        W_fc2 = tf.Variable(tf.truncated_normal( [512, 256*256*2],
                    stddev=0.1, dtype=tf.float32), name='W_fc2' )
        h_fc2 = tf.matmul( h_fc1, W_fc2 )
        h_fc2_re = tf.reshape( h_fc2, [-1, 256, 256, 2] )

    return h_fc2_re
```

TensorBoard는 아래 그림과 같이 신경망을 표시합니다. 가장자리를 자세히 살펴보면 tensor 치수가 적혀 있는 것을 볼 수 있습니다. 즉, 노드 간 이동에 따라, 그래프 전반에서 데이터(tensor)와 데이터 사이즈가 어떻게 변하는지를 확인할 수 있습니다.



실습 1 – 하나의 히든 레이어



우리의 첫번째 실습과제는 한 개의 히든 레이어를 가진, 완전히 연결된 신경망을 만들고, 학습 시키고, 평가하는 것입니다. 신경망에 대한 입력은 각 픽셀의 값, 즉 크기가 256 x 256 (또는 65,536)이 됩니다. 히든 레이어는 사용자가 조정할 수 있는 크기를 가지며 출력은 256 x 256 x 2의 배열이 됩니다. 즉, 각 입력 픽셀은 두 클래스 중 하나에 있을 수 있으므로 픽셀과 연결된 출력 값이 해당 픽셀이 될 수 있습니다. 우리의 경우, 두 클래스는 LV인지 아닌지를 나타냅니다. 손실은 간단히 한번의 함수 호출로 교차 엔트로피 계산과 softmax를 결합하는 `sparse_softmax_cross_entropy_with_logits` (https://www.tensorflow.org/versions/r0.12/api_docs/python/nn.html#sparse_softmax_cross_entropy_with_logits) 라는 TensorFlow 함수를 통해 계산됩니다.

학습 시키기

첫 번째 실습에서 이미 코드가 작성되어 있습니다. 신경망을 학습시키기 위해, 아래를 실행하십시오.

In [4]:

```
!python exercises/simple/runTraining.py --data_dir /dli/data/img_segmentation --checkpoint_dir /dl
```

```
DEBUG images shape inference (1, 256, 256)
DEBUG images shape inference (1, 256, 256)
DEBUG W_fc shape (65536, 512)
DEBUG flatten1_op shape (1, 65536)
DEBUG h_fc1 shape (1, 512)
DEBUG W_fc2 shape (512, 131072)
DEBUG h_fc2 shape (1, 131072)
DEBUG h_fc2_re shape (1, 256, 256, 2)
DEBUG logits shape before (1, 256, 256, 2)
DEBUG labels shape before (1, 256, 256)
DEBUG logits shape after (1, 256, 256, 2)
DEBUG labels shape after (1, 256, 256)
DEBUG cross_entropy shape (1, 256, 256)
OUTPUT: Step 0: loss = 5.600 (0.219 sec)
OUTPUT: Step 100: loss = 4.327 (0.184 sec)
OUTPUT: Step 200: loss = 3.899 (0.183 sec)
OUTPUT: Done training for 1 epochs, 231 steps.
```

모든 것이 제대로 작동하면 화면에 일부 메시지가 인쇄되어 표시될 수 있습니다. 이러한 메시지 중 일부는 TensorFlow에서 보내는 정보 메시지로 일반적으로 무시가 가능합니다. 매 100단계마다 계산되는 손실과 같이 특정 시점에 정보를 출력하기 위해 프로그램에 삽입하여 놓은 "OUTPUT"으로 시작하는 라인을 찾아봅시다. 매우 마지막 라인에서 여러분은 다음과 같은 메시지를 보게 될 것입니다.

```
OUTPUT: Done training for 1 epochs, 231 steps .
```

이것은 한번의 에포크(epoch) 동안 모든 데이터가 학습되어, 여러분이 실습과정 중 하나를 마쳤다는 것을 의미합니다.

평가하기

일단 학습된 모델을 갖게 되면, 여러분은 그것이 처음 접한 데이터에 얼마나 잘 작용하는지 알아보고 싶을 것입니다. 학습된 모델을 평가하기 원한다면, 아래를 실행하십시오.

In [5]:

```
!python exercises/simple/runEval.py --data_dir /dli/data/img_segmentation --checkpoint_dir /dli/ta
        --eval_dir /dli/tasks/tensorboard/eval
```

```
DEBUG images shape inference (1, 256, 256)
DEBUG images shape inference (1, 256, 256)
DEBUG W_fc shape (65536, 512)
DEBUG flatten1_op shape (1, 65536)
DEBUG h_fc1 shape (1, 512)
DEBUG W_fc2 shape (512, 131072)
DEBUG h_fc2 shape (1, 131072)
DEBUG h_fc2_re shape (1, 256, 256, 2)
DEBUG logits eval shape before (1, 256, 256, 2)
DEBUG labels eval shape before (1, 256, 256)
DEBUG logits_re eval shape after (65536, 2)
DEBUG labels_re eval shape after (65536,)
DEBUG correct shape (65536,)
OUTPUT: 2020-02-16 13:32:13.422976: precision = 0.503
OUTPUT: 26 images evaluated from file /dli/data/img_segmentation/val_images.tfrecord
s
```

다시한번, 여러분은 “OUTPUT"으로 시작하는 라인에 주의를 집중하면서 대부분의 TensorFlow 출력을 무시하십시오. 여러분들은 이것을 실행했을 때, 다음과 비슷한 결과를 얻어야 합니다.

```
OUTPUT: 2017-01-26 17:12:28.929741: precision = 0.503
OUTPUT: 26 images evaluated from file /dli/data/img_segmentation/val_images.tfrecords
```

최종 OUTPUT 라인은 모델의 정확도를 나타냅니다. 즉, 모델이 각 픽셀이 LV인지 아닌지를 예측한 정확도를 나타냅니다. 위의 경우 정확성(precision)이 0.503으로, 확률로는 50.3%의 정확도로 클래스를 예측했습니다. 이것은 좋은 결과는 아니지만, 우리가 아주 간단한 네트워크를 사용하여, 단 한번의 이포크(epoch) 동안에만 학습을 실행했다는 것을 고려하면 나쁜 수치는 아닙니다.

TensorBoard

여기에서 TensorBoard 실행하기 (/tensorboard/)

TensorBoard에는 인상적인 시각화 기능이 많이 있습니다. 상단 메뉴에는 "Scalars"라는 링크가 있으며, 이 링크를 클릭하여 캡처 된 일부 정보를 표시할 수 있습니다. 클릭하면 이러한 데이터 중 하나를 확장하여 해당 데이터의 구성을 볼 수 있습니다.

또 다른 메뉴 선택은 "그래프"로, 학습 및 평가 데이터 흐름 그래프를 모두 볼 수 있습니다. 그래프의 각 노드를 클릭하여 확장하면 해당 노드에 대한 자세한 정보를 얻을 수 있으며, 페이지 왼쪽 위에서 작은 드롭다운을 선택하면, 학습 그래프를 볼 것인지 평가 그래프를 볼 것인지를 선택할 수 있습니다.

이 작업에 대한 코드 솔루션은 다음과 같습니다.

```
    with tf.name_scope('Hidden1'):
        W_fc = tf.Variable(tf.truncated_normal( [256*256, 512],
                    stddev=0.1, dtype=tf.float32), name='W_fc')
        flatten1_op = tf.reshape( images_re, [-1, 256*256])
        h_fc1 = tf.matmul( flatten1_op, W_fc )

    with tf.name_scope('Final'):
        W_fc2 = tf.Variable(tf.truncated_normal( [512, 256*256*2],
                    stddev=0.1, dtype=tf.float32), name='W_fc2' )
        h_fc2 = tf.matmul( h_fc1, W_fc2 )
        h_fc2_re = tf.reshape( h_fc2, [-1, 256, 256, 2] )

    return h_fc2_re
```

일부 TensorFlow API 호출과 함께 Python 구문이 사용되는 것을 보실 수 있습니다.

- `tf.name_scope()` 을 사용하면 프로그램의 특정 범위에 이름을 지정할 수 있습니다. 이 기능은 코드를 구성하고 TensorBoard 그래프의 각 노드에 이름을 지정하는 데 유용합니다.
- `tf.Variable()` 은 학습시킬 TensorFlow 변수를 의미하며, 즉 가중치 tensor입니다.
- `tf.reshape()` 은 곧 수행할 작업에 적합한 형태로 tensor를 재형성 하기 위한 TensorFlow 보조 함수입니다.
- `tf.matmul()` 은 여러분이 예상한 것과 같이, 2개의 TensorFlow tensor들을 곱한 메트릭스입니다.

학습 가중치를 의미하는 tf.Variable 텐서(tensor)의 생성이나, 가중치와 입력을 결합하여 신경망을 구축하는 tf.matmul() 함수 작업 등과 같이, 모든 작업들은 명시적으로 호출되는 것을 볼 수 있습니다. 이렇게 하면 원하는 유형의 신경망을 정확하게 정의할 수 있는 유연성이 최대화되지만, 일단 활성화 기능, 바이어스 등을 추가하기 시작하면 비교적 간단한 신경망도 매우 복잡해질 수 있습니다.

이러한 작업을 완화하기 위해 TensorFlow는 그러한 activation, bias, 정규화 등의 기능이 내장된 더 높은 수준의 레이어 API (https://www.tensorflow.org/tutorials/layers)를 가지고 있기 때문에, 자주 쓰이는 convolution, pooling, fully-connected 등의 레이어 타입들을 보완할 수 있습니다. Layers API를 사용하면 TensorFlow에서 매우 간결하게 신경망을 구축할 수 있습니다. 다음 실습에서는 Layers API를 사용하여 신경망을 구축해 보겠습니다.

자세히 다루어 지지 않은 주제들

우리는 완벽한 교육을 위해서는 다루어야 할 많은 주제들을 자세히 논의하지는 못했습니다.

- 모든 데이터가 이미 설정되었다고 가정했습니다. 앞서 보여드린 바와 같이 우리는 이미 설정완료 된 TFRecords 파일 데이터 형식을 사용하고 있습니다.
- 여러 개의 스레드를 사용하여 이러한 파일들의 데이터를 읽어 들이는 TensorFlow 메커니즘을 사용하고 있습니다. 이를 통해 내장된 TensorFlow 기능들을 사용하여 데이터를 랜덤화 할 수 있을 뿐만 아니라 batch_size 및 num_epochs와 같은 것들도 자동 처리할 수 있습니다.
- 데이터 흐름 그래프를 통해 실제 모델 구성에 대한 간략한 설명만 제공했습니다. 여러분이 원하는 경우 확인할 수는 있습니다만, 코드에서 볼 수 있는 Python 구문은 매우 많습니다.
- 마지막으로 구성된 데이터를 쉽게 확인해 볼 수 있도록 TensorBoard로 내보내기 위해, 특수 API 호출을 삽입하였습니다. 이것 또한 여러분이 원한다면 확인하여 볼 수 있는 표준 Python 코드입니다.

실습 2 -- CNN (콘볼루션 신경망)

두 번째 실습은 여러분의 모델을 위에서 보다 더 많은 레이어들과 유형들을 포함하는 더 정교한 네트워크로 전환하는 것입니다. 위의 실습에서는 각각의 개별 픽셀에 초점을 맞추고 있었기 때문에, 관심 영영이 단일 픽셀보다 클 수 있다는 사실을 설명하기 힘들었지만, 우리는 작은 크기의 관심 영역 단위로도 캡처할 수 있게 되기를 원하기 때문에, 앞으로 더 큰 수용 영역을 캡처할 수 있는 콘볼루션 레이어를 활용해 보겠습니다.

또한 대부분의 정보들을 저장하는 동안 데이터를 다운 샘플링 하는 풀링(pooling) 레이어를 더해 컴퓨팅의 복잡성을 제거해 보겠습니다.

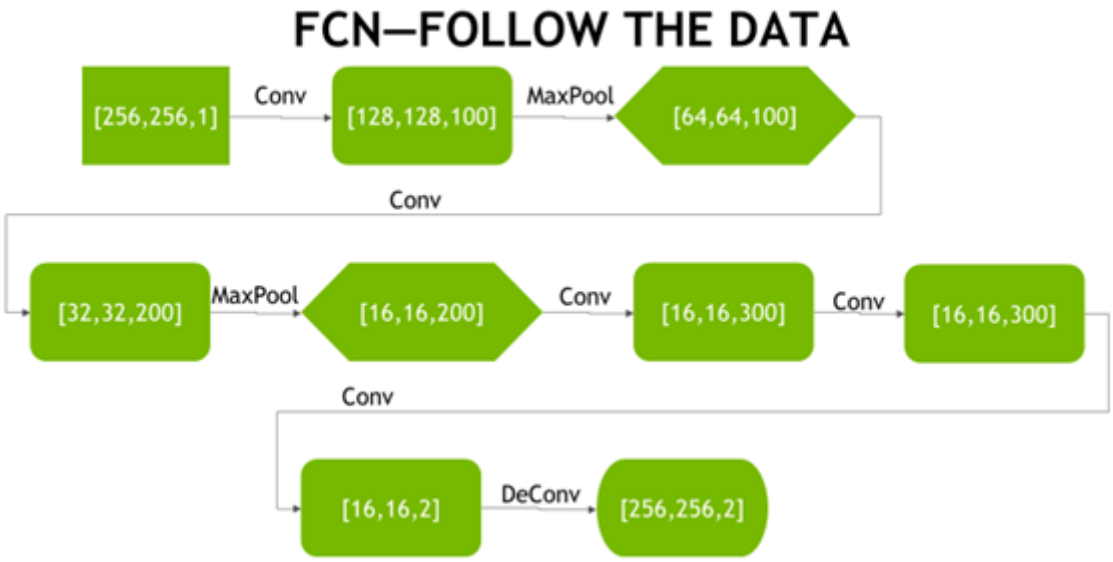
지금까지 우리는 출력 노드의 수가 클래스의 수와 동일한 이미지 인식 신경망과 일반적으로 관련된 레이어를 설명했습니다. 하지만, 우리가 이미지를 분류하는 것 이상의 것을 수행하고 있음을 상기하십시오. 여러분은 이미지의 각 픽셀을 분류하고 있으므로 출력 크기는 클래스 수 (2)와 픽셀 수 (256 x 256)의 곱입니다. 또한 출력 노드의 공간적인 위치도 중요합니다. 각 픽셀은 LV의 일부이거나 그렇지 않은 관련 확률을 갖기 때문입니다.

CNN은 이미지 인식이나 분류 작업에 아주 잘 적용됩니다. 이번 실습에서는 분할(segmentation) 작업을 할 예정인데, 이 것 또한 어떤 의미에서는 분류와 관련이 있습니다. 우리는 이미지 전체를 한꺼번에 분류하는 것이 아니라, 이미지의 각 픽셀들을 분류할 것입니다. 그렇다면, 여기서 문제는 이미지 인식을 잘하는 CNN을 세분화 작업에도 활용할 수 있느냐는 것입니다. 이를 위해서 CNN 모델에 약간의 수정을 해야 할 것으로 보입니다.

표준 이미지 인식 신경망을 사용하여, 완전히 연결된 레이어를 디콘볼루션 레이어 (더 정확하게는 전치 합성곱층 transpose convolution layer (https://www.tensorflow.org/api_docs/python/tf/nn/conv2d_transpose)이라고 불리는)로 대체하면 됩니다.

디콘볼루션은 더 작은 이미지 데이터셋을 최종 픽셀 분류를 위해 원래 크기로 되돌리는 업샘플링 방법입니다. 이 주제에 대해 권해주고 싶은 몇 가지 유용한 참고 자료가 아래 참조 [4, 5, 6] 입니다. 분할작업을 위해 CNN을 수정하여 생성된 신경망을 완전한 콘볼루션 네트워크 (FCN – Fully convolutional network)라고 부릅니다.

이것은 입력 데이터(이 경우 256 x 256 x 1 크기의 tensor)가 그래프를 통해 어떻게 "변화"하는지, 즉 데이터가 이러한 다양한 회전, 풀링 및 작동을 통해 어떻게 변환되는지 시각화 하는 데 도움이 될 수 있습니다. 아래 수치는 다음 작업에서 데이터가 겪게 될 변화를 나타냅니다.



위의 figure에서 보이는 네트워크는 참조 7에서 볼 수 있는 네트워크와 유사합니다. 이 네트워크는 이미지에 표시된 것과 같이 변환된 입력 이미지를 갖는 콘볼루션 레이어, 풀링 레이어 및 최종 디콘볼루션 레이어로 구성되어 있습니다.

이 작업을 수행하려면 이 신경망을 종료한 다음, 그 다음 실습을 실행해야 합니다. 이렇게 하려면 exercises/cnn/neuralnetwork.py (../../../../edit/tasks/task1/task/exercises/cnn/neuralnetwork.py) 파일을 편집하여 모든 `FIXME` 항목들을 코드로 수정하십시오. 각 코드는 여러분의 이해를 돕기 위한 코멘트가 적혀 있으며, 다음 네트워크 구조를 사용하는 것도 도움이 될 것입니다. 레이어들의 이름은 코드를 검사하고, 완료할 때 더 이해하기 쉬울 것입니다.

1. Convolution1, 5 x 5 kernel, stride 2

2. Maxpooling1, 2 x 2 window, stride 2
3. Convolution2, 5 x 5 kernel, stride 2
4. Maxpooling2, 2 x 2 window, stride 2
5. Convolution3, 3 x 3 kernel, stride 1
6. Convolution4, 3 x 3 kernel, stride 1
7. Score_classes, 1x1 kernel, stride 1
8. Upscore (deconvolution), 31 x 31 kernel, stride 16

여러분의 작업을 확인하려면 exercise_solutions/cnn/neuralnetwork.py (../../../../edit/tasks/task1/task/exercise_solutions/cnn/neuralnetwork.py).에서 솔루션을 확인할 수 있습니다.

코드가 완료되면 아래 박스 안의 내용을 사용하여 학습을 시작한 후, 이전 실습에서 열어보았던 TensorBoard 브라우저에서 여러분의 결과물들을 확인해 볼 수 있습니다. 결과가 즉시 보이지 않는다면, TensorBoard가 시각화해야 할 새로운 그래프가 있다는 것을 인식할 때까지 잠시 더 기다려야 하거나, 브라우저를 *새로 고침* 해야 할 수도 있습니다.

In [6]:

```
!python exercises/cnn/runTraining.py --data_dir /dli/data/img_segmentation --checkpoint_dir /dli/t
        --num_epochs 1
```

```
DEBUG images shape inference (1, 256, 256)
DEBUG images shape inference (1, 256, 256)
Traceback (most recent call last):
  File "exercises/cnn/runTraining.py", line 138, in <module>
    tf.app.run()
  File "/usr/local/lib/python2.7/dist-packages/tensorflow/python/platform/app.py", l
ine 126, in run
    _sys.exit(main(argv))
  File "exercises/cnn/runTraining.py", line 135, in main
    run_training()
  File "exercises/cnn/runTraining.py", line 56, in run_training
    results = nn.inference(images)
  File "/dli/tasks/task1/task/exercises/cnn/neuralnetwork.py", line 125, in inferenc
e
    kernel_size=[FIXME,FIXME],
NameError: global name 'FIXME' is not defined
```

학습이 완료된 후 다음을 실행하여 모델이 얼마나 정확한지 확인하십시오.

In [7]:

```
!python exercises/cnn/runEval.py --data_dir /dli/data/img_segmentation --checkpoint_dir /dli/tasks
       --eval_dir /dli/tasks/tensorboard/eval
```

```
DEBUG images shape inference (1, 256, 256)
DEBUG images shape inference (1, 256, 256)
Traceback (most recent call last):
  File "exercises/cnn/runEval.py", line 144, in <module>
    tf.app.run()
  File "/usr/local/lib/python2.7/dist-packages/tensorflow/python/platform/app.py", l
ine 126, in run
    _sys.exit(main(argv))
  File "exercises/cnn/runEval.py", line 141, in main
    run_eval()
  File "exercises/cnn/runEval.py", line 66, in run_eval
    logits = nn.inference(images)
  File "/dli/tasks/task1/task/exercises/cnn/neuralnetwork.py", line 125, in inferenc
e
    kernel_size=[FIXME,FIXME],
NameError: global name 'FIXME' is not defined
```

위에서 실행한 runTraining.py 명령에서 몇 개의 명령을 추가하여 서로 다른 학습 파라메터들을 테스트할 수 있습니다. 시간이 있을 때, --num_epochs 인자를 사용해 보고, 이것이 학습의 정확도에 어떤 영향을 미치는지 확인해 보십시오.

다음은 사용 가능한 명령들의 목록입니다.

```
옵션 목록 :
-h, --help            도움말 메시지를 표시하고 종료합니다.
  --learning_rate LEARNING_RATE
                        학습 속도.
  --decay_rate DECAY_RATE
                        학습 속도 저하률.
  --decay_steps DECAY_STEPS
                        각 학습 속도별 단계.
  --num_epochs NUM_EPOCHS
                       학습 수행 에포크(epoch) 횟수.
  --data_dir DATA_DIR
                       학습데이터를 포함한 디렉토리.
  --checkpoint_dir CHECKPOINT_DIR
                       모델 체크포인트를 사용할 디렉터리.
```

NOTE: 소스 코드를 검사하면 배치 크기를 변경하는 옵션도 볼 수 있습니다. 이번 실습의 목적을 위해 배치 크기를 1로 설정해 주십시오.

```
OUTPUT: 2017-01-27 17:41:52.015709: precision = 0.567
```

에포크를 30으로 늘리면 다음과 같이 훨씬 높은 정확도를 얻게 됩니다.

```
OUTPUT: 2017-01-27 17:47:59.604529: precision = 0.983
```

이와 같이, 학습 기간을 늘리면 정확도가 크게 향상됩니다. 사실 98.3%의 정확도는 매우 높은 것입니다. 이로 만족하나요? 이제 끝난 걸까요?

정확도 (Accuracy)

정확성에 대한 논의의 일환으로, 우리가 정확성을 체크할 때에는, 우리가 정확히 무엇을 컴퓨팅하고 있는지 한걸음 뒤로 물러나 신중히 생각해보아야 할 필요가 있습니다. 현재 우리의 정확도 수치는 간단히 말해 우리가 올바르게 컴퓨팅하고 있는 픽셀이 몇 개나 되는지를 보여주는 것입니다. 그래서 위의 30 이포크의 학습을 한 경우에는, 98.3% 정확하게 픽셀 값을 예측해 냈습니다. 하지만, 좌심실(LV)이 차지하는 영역은 전체 이미지의 크기에 비해 일반적으로 매우 작습니다. 이러한 경우, 클래스 불균형이라고 불리는 문제를 야기하는데, 즉, 한 클래스에 속할 가능성이 다른 클래스에 속할 가능성보다 훨씬 더 높을 수 있다는 것입니다. 단순히 신경망에서 출력되는 모든 픽셀에 대해 LV가 아닌 클래스로 출력했다고 해도, 여전히 95%의 정확도를 가지게 됩니다. 하지만 그러한 신경망은 쓸모 없습니다. 우리는 불균형 이슈와 상관없이, 신경망이 얼마나 잘 LV인지 아닌지를 분류해 낼 수 있는지를 알려주는 정확도 메트릭스가 필요합니다.

실습 3 -- Dice Metric

여러분의 네트워크가 LV를 더 정확하게 분할해 내기 위해 사용할 수 있는 메트릭(Metric) 중 하나는 Dice metric 또는 Sorensen-Dice (https://zetawiki.com/wiki/S%C3%B8rensen%E2%80%93Dice_%EA%B3%84%EC%88%98) 계수라고 불리는 메트릭으로, 두 표본의 유사성을 비교하는 메트릭입니다. 여러분은 우리의 두 관심 영역인, 전문적으로 라벨링 된 영역과 우리의 예측 영역을 비교하는 데 사용할 것입니다. Dice Metric 두 표본의 유사성을 비교하는 Dice metric의 공식은 다음과 같습니다.

$$\frac{2A_{nl}}{A_n + A_l}$$

여기서 A_n은 신경망에 해 예측되는 영역이며, A_l은 전문 분할 레이블에서 얻은 영역이고, A_{nl}은 두 표본의 교차 영역, 즉 네트워크에 의해 올바르게 예측되는 영역으로, 1.0이 만점입니다.

이 메트릭은 클래스 불균형 문제가 제거되었기 때문에, 네트워크가 LV를 얼마나 잘 분할하고 있는지 더 정확하게 계산할 것입니다. 특정 영역 안에 포함된 면적의 크기를 확인하고자 하기 때문에, 여러분은 단순히 주어진 영역의 픽셀 수를 세면 됩니다.

Dice metric이 계산에 얼마나 도움이 되는지 알아보려면 소스 코드 파일 neuralnetwork.py (../../../../edit/tasks/task1/task/exercises/cnnDice/neuralnetwork.py)을 참조하십시오.

1 에포크 동안 아래 셀의 내용을 수행하여 '학습'을 실행 한 후에, 두개 셀 아래의 '평가'를 수행하여 정확도를 확인하십시오. 그 다음, 30 에포크 동안 학습을 실행해 보십시오. 이전에 작업했던 내용과 비슷합니다. 30 이포크의 학습 후에 다시 정확성을 확인해 보십시오. TensorBoard에서 결과를 시각화 하면 됩니다.

In [8]:

```
!python exercises/cnnDice/runTraining.py --data_dir /dli/data/img_segmentation --checkpoint_dir /d
        --num_epochs 1
```

```
DEBUG images shape inference (1, 256, 256)
DEBUG images shape inference (1, 256, 256)
DEBUG conv1 shape (1, 128, 128, 100)
DEBUG pool1 shape (1, 64, 64, 100)
DEBUG conv2 shape (1, 32, 32, 200)
DEBUG pool2 shape (1, 16, 16, 200)
DEBUG conv3 shape (1, 16, 16, 300)
DEBUG conv4 shape (1, 16, 16, 300)
DEBUG drop shape (1, 16, 16, 300)
DEBUG score_classes shape (1, 16, 16, 2)
DEBUG upscore shape (1, 256, 256, 2)
DEBUG logits shape before (1, 256, 256, 2)
DEBUG labels shape before (1, 256, 256)
DEBUG logits shape after (1, 256, 256, 2)
DEBUG labels shape after (1, 256, 256)
OUTPUT: Step 0: loss = 0.825 (0.174 sec)
OUTPUT: Step 100: loss = 0.693 (0.116 sec)
OUTPUT: Step 200: loss = 0.685 (0.114 sec)
OUTPUT: Done training for 1 epochs, 231 steps.
```

In [9]:

```
!python exercises/cnnDice/runEval.py --data_dir /dli/data/img_segmentation --checkpoint_dir /dli/t
       --eval_dir /dli/tasks/tensorboard/eval
```

```
DEBUG images shape inference (1, 256, 256)
DEBUG images shape inference (1, 256, 256)
DEBUG conv1 shape (1, 128, 128, 100)
DEBUG pool1 shape (1, 64, 64, 100)
DEBUG conv2 shape (1, 32, 32, 200)
DEBUG pool2 shape (1, 16, 16, 200)
DEBUG conv3 shape (1, 16, 16, 300)
DEBUG conv4 shape (1, 16, 16, 300)
DEBUG drop shape (1, 16, 16, 300)
DEBUG score_classes shape (1, 16, 16, 2)
DEBUG upscore shape (1, 256, 256, 2)
DEBUG logits eval shape before (1, 256, 256, 2)
DEBUG labels eval shape before (1, 256, 256)
DEBUG logits_re eval shape after (65536, 2)
DEBUG labels_re eval shape after (65536,)
DEBUG labels_re eval shape after (65536, 1)
DEBUG indices shape (65536, 1)
DEBUG example_sum shape ()
DEBUG label_sum shape ()
DEBUG sum_tensor shape (65536, 1)
DEBUG twos shape (65536, 1)
DEBUG divs shape (65536, 1)
OUTPUT: 2020-02-16 13:32:48.268762: Dice metric = 0.033
OUTPUT: 26 images evaluated from file /dli/data/img_segmentation/val_images.tfrecord
s
```

만약 여러분이 1 에포크로 학습을 진행했다면, 아마도 1% 미만의 정확도를 얻었을 것입니다. 앞서 수행에서 아래와 같이 1 이포크 후의 수행결과는

```
OUTPUT: 2017-01-27 18:44:04.103153: Dice metric = 0.034
```

30 에포크의 학습 수행 이후에는 약 57%의 정확도를 갖습니다.

```
OUTPUT: 2017-01-27 18:56:45.501209: Dice metric = 0.568
```

보다 사실적인 정확도 측정법을 사용하면, 신경망을 더 개선할 수 있다는 것을 알 수 있습니다.

파라미터 검색(Parameter Search)

학습 속도(learning rate)는 우리가 예측을 다시 실행할 때마다 가중치가 조정되는 비율을 의미합니다. 만약 이 값이 너무 크면, 우리는 결국 너무 큰 값에 의해 가중치들을 조정하게 될 것이고, 바람직한 솔루션으로 수렴해 가는 대신, 그 주변에서 오락가락할 수도 있습니다. 반대로 학습 속도 값이 너무 작다면, 가중치에 대한 조정치가 너무 작아서, 우리가 만족할 만한 솔루션이 되기까지 너무 오랜 시간이 소요될 수도 있습니다. 자주 이용되는 기법은 변수 또는 조정 가능한 학습 속도의 도입입니다. 학습을 시작할 때, 우리는 좀 큰 수치의 학습 속도 값을 사용하여, 기대하는 솔루션에 가까운 결과에 더 빠르게 도달하게 되기를 희망해보겠습니다. 그런 다음, 학습이 더욱 진행됨에 따라, 우리가 원하는 솔루션에 도달할 때까지 학습 속도 값을 연속적으로 줄여보겠습니다. 위에 명시된 세 가지 파라미터들은 학습 속도와 학습 속도의 변화정도와 변화 빈도를 제어하는데 도움이 됩니다. 기본적으로, 이러한 옵션들을 변경하지 않았을 경우 사용되는 기본값은 다음과 같습니다.

```
--learning_rate 0.01
--decay_rate 1.0
--decay_step 1000
--num_epochs 1
```

다음 셀의 학습을 수행하면서 이러한 수치들을 변경해 가면서, 이전보다 더 높은 정확도를 도출할 수 있는지 실험해 보십시오. 안타깝게도 실습 시간의 제약 때문에 100 에포크 이상을 수행하기를 권하지는 않습니다만, 실제 현장에서는 훨씬 더 많은 횟수의 에포크를 시도해 보기를 바랍니다.

학습이 시작된 이후 에포크의 회수를 너무 많이 설정했다는 것을 깨닫게 되더라도 실망하지 마십시오. 여러분들은 여전히 학습을 정지하고, 모델을 테스트하고, 평가할 수 있습니다. (아래 두 단계 밑의 셀 참조) TensorFlow에는 모델을 정기적으로 스냅샷하는 체크포인트 기능이 있으므로, 여러분이 학습 실행을 중지한 후에도 가장 최근의 스냅샷은 저장되어 있습니다.

In [10]:

```
!python exercises/cnnDice/runTraining.py --data_dir /dli/data/img_segmentation  --checkpoint_dir /
       --num_epochs 1 --learning_rate 0.01 --decay_rate 1.0 --decay_steps 1000
```

```
DEBUG images shape inference (1, 256, 256)
DEBUG images shape inference (1, 256, 256)
DEBUG conv1 shape (1, 128, 128, 100)
DEBUG pool1 shape (1, 64, 64, 100)
DEBUG conv2 shape (1, 32, 32, 200)
DEBUG pool2 shape (1, 16, 16, 200)
DEBUG conv3 shape (1, 16, 16, 300)
DEBUG conv4 shape (1, 16, 16, 300)
DEBUG drop shape (1, 16, 16, 300)
DEBUG score_classes shape (1, 16, 16, 2)
DEBUG upscore shape (1, 256, 256, 2)
DEBUG logits shape before (1, 256, 256, 2)
DEBUG labels shape before (1, 256, 256)
DEBUG logits shape after (1, 256, 256, 2)
DEBUG labels shape after (1, 256, 256)
OUTPUT: Step 0: loss = 0.706 (0.175 sec)
OUTPUT: Step 100: loss = 0.693 (0.118 sec)
OUTPUT: Step 200: loss = 0.693 (0.115 sec)
OUTPUT: Done training for 1 epochs, 231 steps.
```

In [11]:

```
!python exercises/cnnDice/runEval.py --data_dir /dli/data/img_segmentation --checkpoint_dir /dli/t
       --eval_dir /dli/tasks/tensorboard/eval
```

```
DEBUG images shape inference (1, 256, 256)
DEBUG images shape inference (1, 256, 256)
DEBUG conv1 shape (1, 128, 128, 100)
DEBUG pool1 shape (1, 64, 64, 100)
DEBUG conv2 shape (1, 32, 32, 200)
DEBUG pool2 shape (1, 16, 16, 200)
DEBUG conv3 shape (1, 16, 16, 300)
DEBUG conv4 shape (1, 16, 16, 300)
DEBUG drop shape (1, 16, 16, 300)
DEBUG score_classes shape (1, 16, 16, 2)
DEBUG upscore shape (1, 256, 256, 2)
DEBUG logits eval shape before (1, 256, 256, 2)
DEBUG labels eval shape before (1, 256, 256)
DEBUG logits_re eval shape after (65536, 2)
DEBUG labels_re eval shape after (65536,)
DEBUG labels_re eval shape after (65536, 1)
DEBUG indices shape (65536, 1)
DEBUG example_sum shape ()
DEBUG label_sum shape ()
DEBUG sum_tensor shape (65536, 1)
DEBUG twos shape (65536, 1)
DEBUG divs shape (65536, 1)
OUTPUT: 2020-02-16 13:33:19.613949: Dice metric = 0.033
OUTPUT: 26 images evaluated from file /dli/data/img_segmentation/val_images.tfrecord
s
```

우리가 얻은 결과치 중 하나는 86%의 정확성을 갖았습니다. 학습에 사용된 파라미터가 무엇인지 보기 위해 A를 확인하십시오.

추가 개선하기

교육용 실습이었기 때문에, 우리는 주어진 시간 안에 완료 가능한 작은 규모의 작업을 수행해 보았습니다만, 실제 현업에서 이미지 분할 작업을 제대로 수행하려면 어떻게 해야 할까요? 여러분이 개선을 위해 고려해야 할 몇가지들을 소개하겠습니다.

- 더 오래 학습 – 이번 실습에서는 짧은 학습 시간을 실행했지만, 현실에서는 더 많은 에포크 회수를 실행하기 바랍니다.
- 더 많은 학습 데이터 – 학습 데이터로 단 236개의 이미지만 사용했지만, 더 많은 데이터를 모으거나, 이미 가지고 있는 이미지의 회전, 반전 등을 통해 학습 데이터의 개수를 늘릴 필요가 있습니다. 이를 위해 TensorFlow에는 이미지를 자동으로 플립/회전/순서변환 하는 기능이 내장되어 있습니다.
- 더 큰 네트워크 – AlexNet나 다른 대형 CNN을 사용해 보고, 이들을 FCN으로 바꾸어 보기 바랍니다.

요약

이번 실습에서는 TensorFlow를 프레임워크로 선택하여 이미지 분할에 대해 알아보았습니다. 여러분은 표준 CNN을 분할 네트워크로 사용하기 위해 FCN으로 변환하는 법을 배웠습니다. 또한 신경망 학습 시, 올바를 정확도 메트릭을 선택해야 하는 것이 얼마나 중요한지도 확인했습니다. 마지막으로, 파라메터 조정을 수행하는 것이 딥러닝 워크플로우의 필수적인 부분이며, 종국에는 우리의 목표 작업을 수용할 만한 정확도로 수행할 수 있도록 신경망을 안정화하는 데 도움이 되는지를 확인하였습니다.

더 배워봅시다

자세히 알아보려면 다음 자료를 사용하십시오.

- CUDA 개발자 존 (https://developer.nvidia.com/category/zone/cuda-zone)에서 더 알아보십시오.
- 시스템에 NVIDIA GPU가 있으면 CUDA 툴킷 (https://developer.nvidia.com/cuda-toolkit)을 다운받아 설치하십시오.
- 병렬 프로그래밍에 대한 [무료 Udacity 온라인 강좌]((https://www.udacity.com/course/cs344)를 (https://www.udacity.com/course/cs344)%EB%A5%BC) 수강하십시오.
- Cuda 테그를 사용하여 Stackoverflow (http://stackoverflow.com/questions/tagged/cuda)에서 답을 찾거나 질문하십시오.

랩을 마치며

마지막으로, 시간이 다되어 인스턴스가 종료되기 전에 이번 실습에서 작업한 내용을 저장하는 것을 잊지 마십시오.

1. 이 창 위쪽에 있는 `File -> Download as -> IPython(.ipynb)` 으로 이동하여 이 IPython 노트북을 저장합니다.
2. 다음 셀 블록을 실행하여 작업 중인 파일의 zip 파일을 만든 후 아래 링크와 함께 다운로드할 수 있습니다.

In [12]:

```
!tar -czf ImageSegmentation.tar.gz exercise_solutions/ exercises/ images/*png
```

ImageSegmentation.tar.gz (./ImageSegmentation.tar.gz)

Lab FAQ

Q: 셀을 실행하는 데 문제가 있거나 다른 기술적 문제가 있습니까?
A: 인프라 FAQ (https://developer.nvidia.com/self-paced-labs-faq#Troubleshooting)를 참조하십시오.

Q: 작업을 실행할 때 예상치 못한 동작(즉, 잘못된 출력)이 발생합니다.
A: CUDA Runtime API 호출 중 하나 이상이 실제로 오류를 반환하는 것일 수 있습니다. CUDA 런타임 오류에 대한 오류가 화면에 인쇄됩니까?

References

[1] Sunnybrook cardiac images from earlier competition https://smial.sri.utoronto.ca/LV_Challenge/Data.html (https://smial.sri.utoronto.ca/LV_Challenge/Data.html)

[2] This "Sunnybrook Cardiac MR Database" is made available under the CC0 1.0 Universal license described above, and with more detail here: http://creativecommons.org/publicdomain/zero/1.0/ (http://creativecommons.org/publicdomain/zero/1.0/)

[3] Attribution: Radau P, Lu Y, Connelly K, Paul G, Dick AJ, Wright GA. "Evaluation Framework for Algorithms Segmenting Short Axis Cardiac MRI." The MIDAS Journal -Cardiac MR Left Ventricle Segmentation Challenge, http://hdl.handle.net/10380/3070 (http://hdl.handle.net/10380/3070)

[4] http://fcn.berkeleyvision.org/ (http://fcn.berkeleyvision.org/)

[5] Long, Shelhamer, Darrell; "Fully Convoutional Networks for Semantic Segmentation", CVPR 2015.

[6] Zeiler, Krishnan, Taylor, Fergus; "Deconvolutional Networks", CVPR 2010.

[7] https://www.kaggle.com/c/second-annual-data-science-bowl/details/deep-learning-tutorial (https://www.kaggle.com/c/second-annual-data-science-bowl/details/deep-learning-tutorial)

정답

[A] 다음 구성에서는 약 86%의 정확도를 갖습니다.

```
--learning_rate 0.03
--decay_rate 0.75
--num_epochs 100
--decay_steps 10000
OUTPUT: 2017-01-27 20:19:08.702868: Dice metric = 0.862
```

(https://www.nvidia.com/dli)