Deep Traffic Sign Classification using LeNet CNN

Author: Carolina Hoffmann-Becking
Github: carolina-github
Date: 19 Oct. 2020

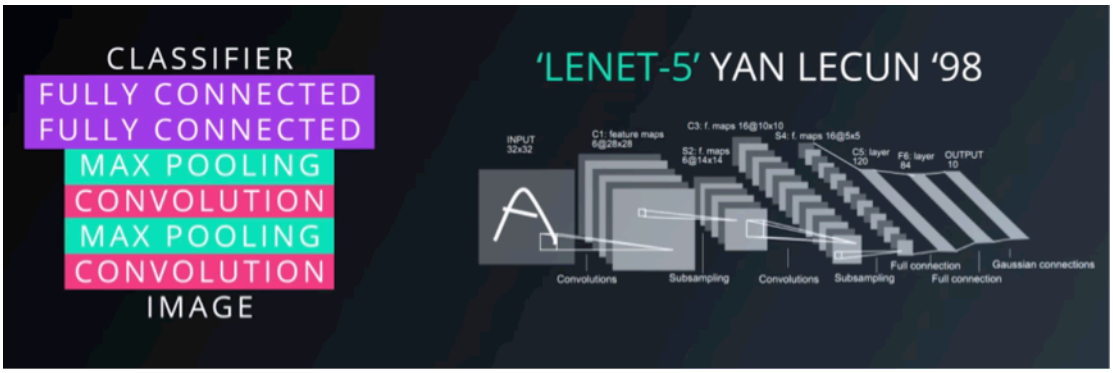
Goal

Design and implement a deep learning model that learns to recognize traffic signs from the German Traffic Sign Dataset

Dataset source: http://benchmark.ini.rub.de/?section=gtsrb&subsection=dataset

Approach

Libraries: pickle, pandas, matpotlib, numpy, tenserflow, sklearn, cv2



Result

95% total accuracy on traffic sign classification on test dataset

1. Dataset Exploration

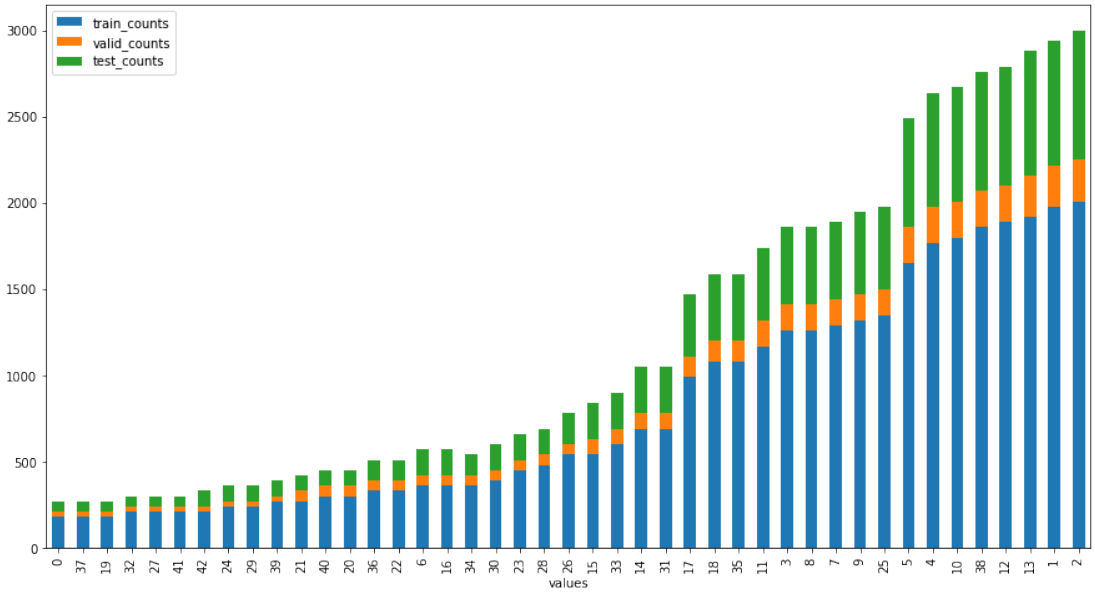
The length of the data dictionary labels key provides information for the respective datasets:

- The size of training set is **34799**
- The size of the validation set is **4410**
- The size of test set is **12630**

The shape of a traffic sign image is **32x32x3** which is the optimal format to feed into the LeNet learning architecture.

I used pandas and numpy to calculate the unique classes in the pandas dataframe labels column, which outputs **43 unique classes**.

The **distribution of labels** can bee seen below split into respective training (blue), validation (orange) and test (green) dataset counts:



Conclusion

The data training the model (training and validation set) represent ~70%, while the testing dataset is ~30%. This is a standard rule of thumb.

The label distribution shows that about half of the traffic signs have <1000 training images. To further boost accuracy across the signs and to minimise bias additional images could be fed into the model for “underrepresented” traffic signs.

2. Design and Test a Model Architecture

2.1 Image preprocessing

Objective

Normalising the image pixels to values with a mean of ~zero and equal variance allows a well conditioned ground for optimization. The optimizer is better conditioned (“less searching”) to find an optimal solution in a balanced universe of values.

Approach

(Pixel - 128)/ 128 is a basic approach to approximately normalize image data. This approach does not change the context of the image.

Result

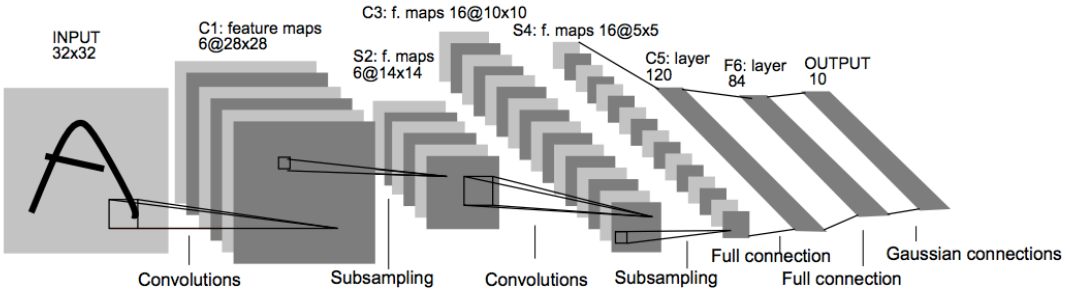
New values of image data range from -1 to 0.99 representing roughly zero mean and equal variance.

2.2 Model Architecture

Background

The Objective is to use different stacked layers of convolutions, max pooling, flatten and fully connected layers to classify traffic sign images with a >93% accuracy.

The learning architecture follows the LeNet architecture from Yan Lecun shown below. The architecture is small and easy to understand, yet large enough to provide interesting results. In this project a test accuracy of 95% has been achieved.



Code

1. First Convolution

In a convolution layer, a filter is applied to the image. The filter strides over the image and outputs a new range of feature maps (k_output) of the image. The filter shares the same weights across the image input.

Input: 32x32x3

- tf.nn.conv2d(image, weights, strides=[1, 1, 1, 1], padding='VALID') + biases
- weights = tf.Variable(tf.truncated_normal(shape=(5, 5, 3, 6), mean = mu, stddev = sigma))
- shape = (filter_size_height, filter_size_width, color_channels, k_output)
- biases = tf.Variable(tf.zeros(6))

Output: 28x28x6 because of stride = 1, padding = Valid and k_ouput defined in weights = 6

--
Explanatory note on formula for calculating the filter size

- Input = [32,32,3], Output = [28,28,6]
- out_height_width = (float(in_height_width - filter_height_width + 1) / float(strides))
- Output[0] = (Input[0] - filter_height_width + 1)/1
- filter_height_width = Input[0]-Output[0]+1 = 32-28+1 = 5
- filter_size_height = filter_size_width

Explanatory note on Initializing weights with tf.truncated_normal

- tf.truncated_normal: the randonly generated values follow a normal distribution with specified mean and standard deviation, except that values whose magnitude is more than 2 standard deviations from the mean are dropped and re-picked. It's called truncated because you are cutting off the tails from a normal distribution.

2. **Relu activation**

Adding additional layers such as the activation layer turns the model into a nonlinear function. This nonlinearity allows the network to solve more complex problems.

- A Rectified linear unit (ReLU) is type of activation function that is defined as f(x) = max(0, x). The function returns 0 if x is negative, otherwise it returns x.
- tf.nn.relu(conv1)

3. **Max Pooling**

Conceptually, the benefit of the max pooling operation is to reduce the size of the input, and allow the neural network to focus on only the most important elements. Max pooling does this by only retaining the maximum value for each filtered area, and removing the remaining values.

Input: 28x28x6

- tf.nn.max_pool(x, ksize=[1, k, k, 1], strides=[1, k, k, 1], padding='VALID' or "SAME)
- tf.nn.max_pool(conv1, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='VALID')

Output: 14x14x6 because of k (the size of the filter) = 2 and stride = 2

--

Explanatory note on Strides

- Stride represents at what steps the filter is run over the image
- stride for each dimension (batch_size, height, width, depth)
- For both ksize and strides, the batch_size and channel_depth dimensions are typically set to 1.

Explanatory note on Padding

- Padding is either 'VALID' or 'SAME'.
- If padding is 'SAME' then the filter will overlap the edges of the image and the image size will remain the same due to zero padding
- If padding is 'VALID' the image size will be smaller, i.e. 32x32 > 28x28 for stride = 1 and valid padding since there are fewer filter operations when not overlapping the four edges of the image

---

4. **Second Convolution [**14x14x16 to 10x10x16]

5. **Relu activation**

6. **Max Pooling** [10x10x16 to 5x5x16]

---

7. **Flattening layer**

Input: 5x5x16

- flatten(conv2)

Output: 400 (5*5*16)

---

8. **First fully connected layer**

Input: 400

- Tf.matmul = Matrix multiplication
- tf.matmul (flattening layer, weights) + biases
- weights = tf.Variable(tf.truncated_normal(shape=(400, 120), mean = mu, stddev = sigma))
- biases= tf.Variable(tf.zeros(120))

Output: 120

--
Explanatory note on Output

- The weights of a matrixmultiplication correspond to the output shape of the layer, which is defined as 400,120 according to the LeNet structure. However, this hyperparameter can be adjusted in the code defining the weights.

9. Relu activation of fully connected layers

- hidden layer = tf.nn.relu(fc1)
- tf.nn.dropout(hidden layer, keep_prob)
- hidden_layer: the tensor to which you would like to apply dropout
- keep_prob: the probability of keeping (i.e. not dropping) any given unit

--
Explanatory note on Dropout Regularisation

- The network thats just the right size for your data is very very hard to optimize. In practice, we always try networks that are way too big for our data and then we try our best to prevent them from overfitting.
- The values that go from one layer to the next are called activations
- Randomly, for every example you train your network on, set a set number of the activations to 0, i.e. keep_prob = 0.5 then set half of the activations to 0
- At the same time factor the remaining activations by a factor of 1/keep_prob
- Take the consenus by averaging the activations

Debugging - Dropout regularisation

- ✓ During training, a good starting value for keep_prob is 0.5.
- ✓ During testing, use a keep_prob value of 1.0 to keep all units and maximize the power of the model.
- ✓ Regularization only applies to the fully-connected region of your convnet, because if you add dropout between conv layers. It'll only degrade the performance further since conv layers are already very sparse.

---

10. Second fully connected layer [120 to 84]

11. Relu activation

---

12. Third fully connected layer [84 to 43]

---

13. Result = Logits (probabilites) for all 43 classes

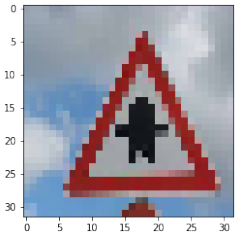
2.3 Visualize layers of the neural network

While neural networks can be a great learning device they are often referred to as a black box. We can understand what the weights of a neural network look like better by plotting their feature maps. After successfully training your neural network you can see what it's feature maps look like by plotting the output of the network's weight layers in response to a test stimuli image.

From these plotted feature maps, it's possible to see what characteristics of an image the network finds interesting. For a sign, maybe the inner network feature maps react with high activation to the sign's boundary outline or to the contrast in the sign's painted symbol.

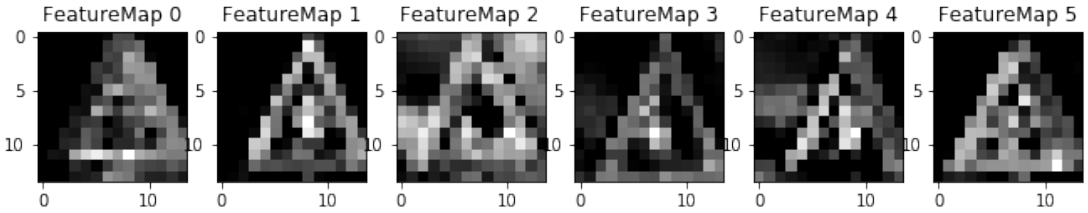
- tf_activation.eval(session=sess,feed_dict={x : image_input})
- tf_activation: should be a tf variable name used during your training procedure that represents the calculated state of a specific weight layer

2.3.1 Image Input



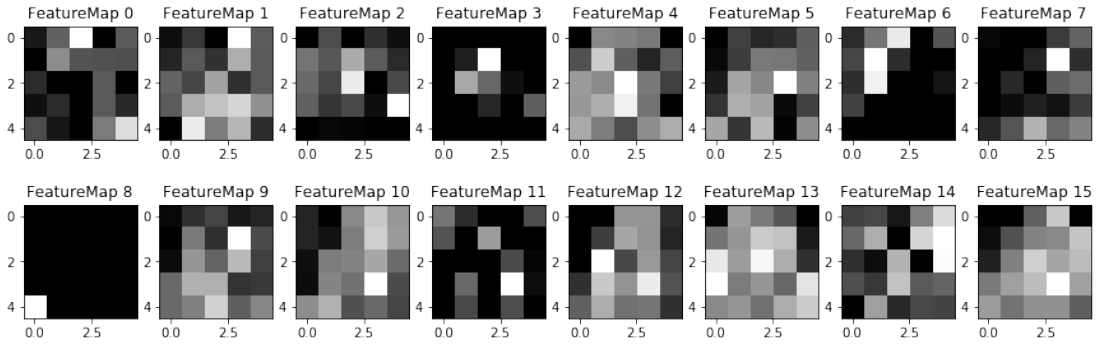
2.3.2 First convolutional layer

- First convolutional layer has k_output = 6 feature maps



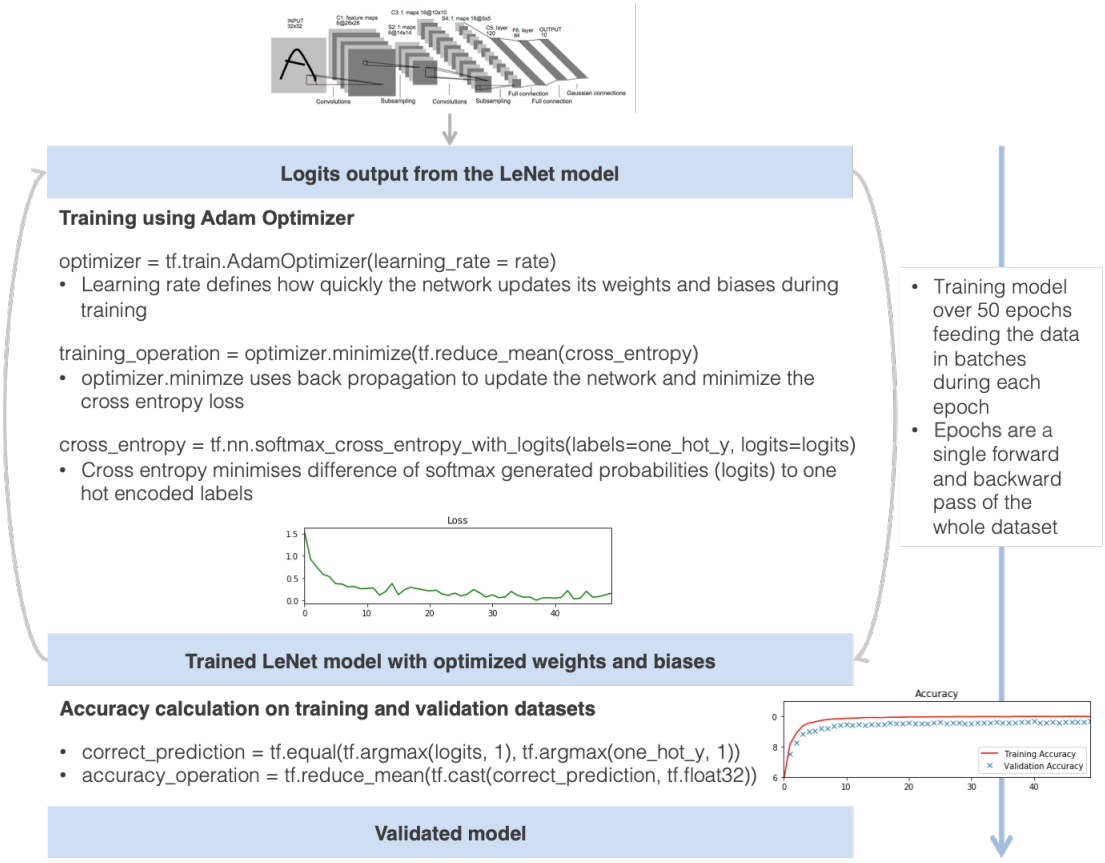
2.3.3 Second convolutional layer

- Second convolutional layer has k_output = 16 feature maps



3. Training

3.1 Approach



Referenced code in Approach chart

- optimizer = tf.train.AdamOptimizer(learning_rate = rate)
- cross_entropy = tf.nn.softmax_cross_entropy_with_logits(labels=one_hot_y, logits=logits)
- training_operation = optimizer.minimize(tf.reduce_mean(cross_entropy))
- correct_prediction = tf.equal(tf.argmax(logits, 1), tf.argmax(one_hot_y, 1))
- accuracy_operation = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))

3.1.1 Adam Optimizer

- AdamOptimizer is more sophisticated than stochastic gradient descent and a good default optimizer Adaptive Gradient Algorithm (AdaGrad) that maintains a per-parameter learning rate that improves performance on problems with sparse gradients (e.g. natural language and computer vision problems).
- The method computes individual adaptive learning rates for different parameters from estimates of first and second moments of the gradients.
- optimizer.minimze uses back propagation to update the network and minimize the training loss
- optimizer = tf.train.AdamOptimizer(learning_rate = rate)
- training_operation = optimizer.minimize(loss_operation)

3.1.2 Learning rate

- For Optimizer (i.e. Gradient Descent, Adam Optimizer) to update weights and bias during training. New weights and bias deduct learning rate * derivative of weights and bias.
- Learning rate defines how quickly the network updates its weights with 0.001 being a good default value
- Stay calm and decrease your learning rate for better accuracy
- rate = 0.001

3.1.3 Cross entropy loss function

- Cross entropy minimises difference of softmax generated probabilities (logits) to one hot encoded labels
- tf.reduce_mean averages the difference from logits to ground truth labels
- cross_entropy = tf.nn.softmax_cross_entropy_with_logits(labels=one_hot_y, logits=logits)
- loss_operation = tf.reduce_mean(cross_entropy)

3.2 Execution

- Training model over 50 epochs feeding the data in batches during each epoch
- sess.run(training_operation, feed_dict={x: X_train, y: y_train})

3.2.1 Epochs and Batch Size

EPOCHS = 50

- The eopch and batch size values affect the training speed and model accuracy
- Epochs are a single forward and backward pass of the whole dataset during training

BATCH_SIZE = 128

- The larger the batch size the faster the model will train, however memory limitations
- Batch size Is the number of datapoints per batch. Number of batches = number of images / batch_size
- Batch x = images, batch y = Labels

3.3 Evaluation

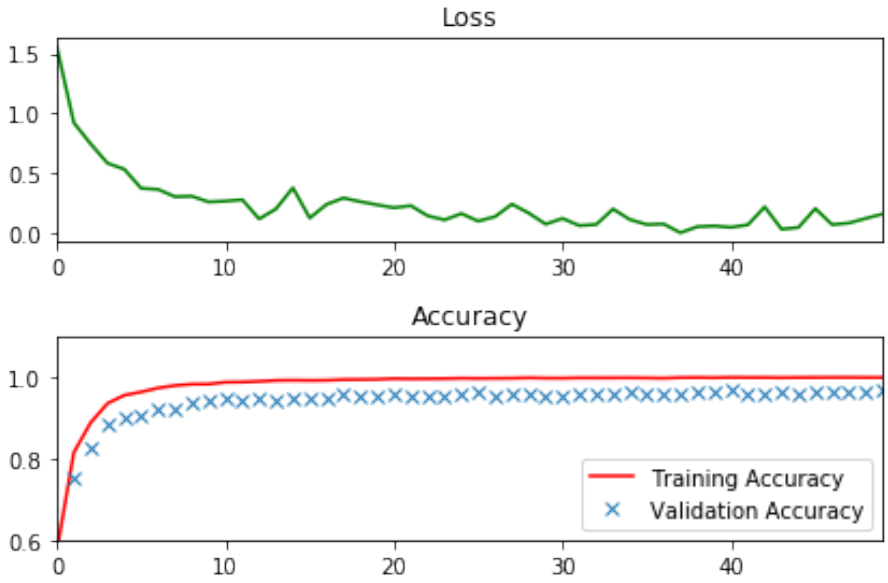
3.3.1 Accuracy calculation

- correct_prediction = tf.equal(tf.argmax(logits, 1), tf.argmax(one_hot_y, 1))
- accuracy_operation = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))

- tf.argmax(logits, 1) outputs the correct label, which is the label with the max probability across the logits
- tf.argmax(one_hot_y, 1) outputs the actual true values
- tf.equal compares the two tensors and returns a list of booleans [True, False, True...] for all the predictions
- Convert (cast) the list of booleans into a list of binary value [0,1,0,..] and calculate the accuracy mean of each batch

4. Interpretation of results

A validation set can be used to assess how well the model is performing. A low accuracy on the training and validation sets imply underfitting. A high accuracy on the training set but low accuracy on the validation set implies overfitting.



Conclusion

According to the chart above the model seems to be slightly overfitting as validation accuracy is always lower than training accuracy. To avoid overfitting images can as an example be converted to grayscale images.

Further methods to improve the model are listed in the final section of this document "areas of improvement".

5. Final Result

My final model results were:

- training set accuracy of 99.9%
- validation set accuracy of 96.7%
- test set accuracy of **95.0%**

Reflection on initial models

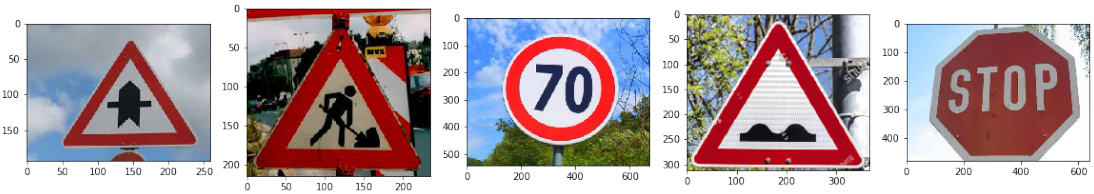
The first architecture chosen had no regularisation of the fully connected layers. With the implementation of dropout regularisation the final model is less likely to overfit.

In addition the training epochs have been increased from originally 10 epochs to 50 epochs to allow longer training and improved optimization of weights and biases.

5. Test model on new images from the web

5.1 Image input

The trained model is tested on five new German Traffic signs found on the web. The images are visualized below.



5.2 Model performance on new images

The new images have been classified by the model with an accuracy of 80%, implying that 4 out of 5 images have been correctly classified.

5.3 Investigation of wrongly classified image

Detailed investigation into the wrongly classified image by looking at individual cross entropy loss values and top 3 softmax probabilites.

5.3.1 Cross entropy loss

- Cross entropy loss = [0.00000000e+00 1.97274055e+01 0.00000000e+00 0.00000000e+00 8.04588199e-03]
- Cross entropy average = 3.94709

Interpretation

- Image 1 and 4 have been classified with absolute confidence since cross entropy, the difference between logits and one hot encoding is 0
- Image 3 and 4 have minor differences / lower confidence
- Image 2 has the highest loss and represents close to the total loss since the average 1.972/5 = 0.3944

Debugging – Output of Cross entropy loss

- ✓ The saved session needs to be restored
- ✓ X, Y and keep prob vales need to be fed into the session via the feed_dict
- ✓ Originally, I had only fed in X values which led to error

Final Solution:

```
with tf.Session() as sess:
    saver.restore(sess, tf.train.latest_checkpoint('.'))
    cross_entropy_new = sess.run(cross_entropy, feed_dict={x: X_new_norm, y: y_new, keep_prob: 1.0})
    cross_entropy_average = sess.run(loss_operation, feed_dict={x: X_new_norm, y: y_new, keep_prob: 1.0})
    print(cross_entropy_new)
    print(cross_entropy_average)
```

5.3.2 Softmax probabilities

- For each of the new images the top 3 softmax probabilites are shown below.
- tf.nn.top_k will return the values and indices (class ids) of the top k predictions.
- A Softmax Regression returns a list of values between 0 and 1 that add up to one

Top 3 softmax probabilites and corresponding labels of new images

- [11, 30, 27], [1.00000000e+00, 3.82318222e-12, 2.73875525e-15]
- [27, 11, 30], [8.41914833e-01, 1.58053279e-01, 2.31592148e-05]
- [4, 8, 0], [1.00000000e+00, 3.19021441e-14, 6.12853140e-20]
- [22, 29, 26], [1.00000000e+00, 1.53058117e-08, 6.35763781e-19]
- [14, 15, 13], [9.91986394e-01, 7.36280158e-03, 6.49597379e-04]

5.3.3 Ground truth of new images

y_new = [11, 25, 4, 22, 14]

- 11 - Right-of-way at the next intersection
- 25 - Road work
- 4 - Speed limit (70km/h)
- 22 - Bumpy road
- 14 - Stop

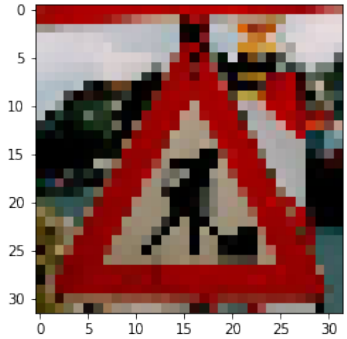
5.4 Interpretation of results

The top 3 softmax probabilities show that the roadwork image has not been correctly classified in neither of the top 3 classes. It has shown a high confidence for Right-of-way at the next intersection. This is expected due to the unclear pixel due to image cropping during data preprocessing.

[27,11,30]

- 27: Pedestrians
- 11: Right-of-way at the next intersection
- 30: Beware of ice/snow

Preprocessed roadwork image for reference



7. Areas of improvement

7.1 Augment the training data

- Augmenting the training set might help improve model performance
- Common data augmentation techniques include rotation, translation, shifting, zoom, flips, and/or color perturbation (i.e. conversion to grayscale). These techniques can be used individually or combined.

7.2 Analyze new image performance in more detail with precision and recall

- Calculate the precision and recall for each traffic sign type from the test set and then compare performance on these five new images
- Looking at performance of individual sign types can help guide how to better augment the data set or how to fine tune the model.
- Interpret the results of precision and recall values: If one of the new images is a stop sign but was predicted to be a bumpy road sign, then we might expect a low recall for stop signs. In other words, the model has trouble predicting on stop signs. If one of the new images is a 100 km/h sign but was predicted to be a stop sign, we might expect precision to be low for stop signs. In other words, if the model says something is a stop sign, we're not very sure that it really is a stop sign.

7.3 Experiment with different network architectures

- Change the dimensions of the LeNet layers, i.e the output size of the fully connected layers
- Change values for biases, since tf.zeros don’t provide any randomness or shift along y axis and hence is not the optimal choice. A random initializer can be used instead
- Add regularization features for the conv layers such as batch normalization between your convolutions. This can regularize the model, as well as make the model more stable during training