**Convolutional Neural Networks for Sentence Classification**

Anita Mohandas mohandas.a@husky.neu.edu

Radhika Shroff Shroff.r@husky.neu.edu

Sindhuri Yamala yamala.s@husky.neu.edu

*Information System, Northeastern University*

**Abstract:**

We report on a series of experiments with convolutional neural networks (CNN) trained on top of pre-trained word vectors for sentence-level classification tasks. We show that a simple CNN with little hyperparameter tuning and static vectors achieves excellent results on multiple benchmarks. Learning task-specific vectors through fine-tuning offers further gains in performance. We additionally propose a simple modification to the architecture to allow for the use of both task-specific and static vectors. The CNN models discussed herein improve upon the state of the art on 4 out of 7 tasks, which include sentiment analysis and question classification.

**1 Introduction:**

Deep learning models have achieved remarkable results in computer vision and speech recognition in recent years. Within natural language processing, much of the work with deep learning methods has involved learning word vector representations through neural language models and performing composition over the learned word vectors for classification. Convolutional Neural network is a type of feed forward artificial neural network in which connectivity pattern between neurons is inspired by the organization of animal visual cortex. Convolutional neural networks (CNN) utilize layers with convolving filters that are applied to local features. To improve their performance, we can collect larger datasets, learn more powerful models, and use better techniques for preventing overfitting. Originally invented for computer vision, CNN models have subsequently been shown to be effective for NLP and have achieved excellent results in semantic parsing, sentence modeling and other traditional NLP tasks.

Despite the attractive qualities of CNNs, and despite the relative efficiency of their local architecture, they have still been prohibitively expensive to apply in large scale to high-resolution images. Luckily, current GPUs, paired with a highly-optimized implementation of 2D convolution, are powerful enough to facilitate the training of interestingly-large CNNs, and recent datasets such as ImageNet contain enough labeled examples to train such models without severe overfitting.

In the present work, we train a simple CNN with one layer of convolution on top of word vectors obtained from an unsupervised neural language model. These vectors were trained by Mikolov et al on 100 billion words of Google News, and are publicly available.1 We initially keep the word vectors static and learn only the other parameters of the model. Despite little tuning of hyperparameters, this simple model achieves excellent results on multiple benchmarks, suggesting that the pre-trained vectors are ‘universal’ feature extractors that can be utilized for various classification tasks. Learning task-specific vectors through fine-tuning results in further improvements. We finally describe a simple modification to the architecture to allow for the use of both pre-trained and task-specific vectors by having multiple channels.

Our work is philosophically like Yoon Kim’s work for sentence classification for convolutional neural network and r to Razavian which showed that for image classification, which showed that for image classification, feature extractors obtained from a pretrained deep learning model perform well on a variety of tasks—including tasks that are very different from the original task for which the feature extractors were trained.

**2 CNN:**

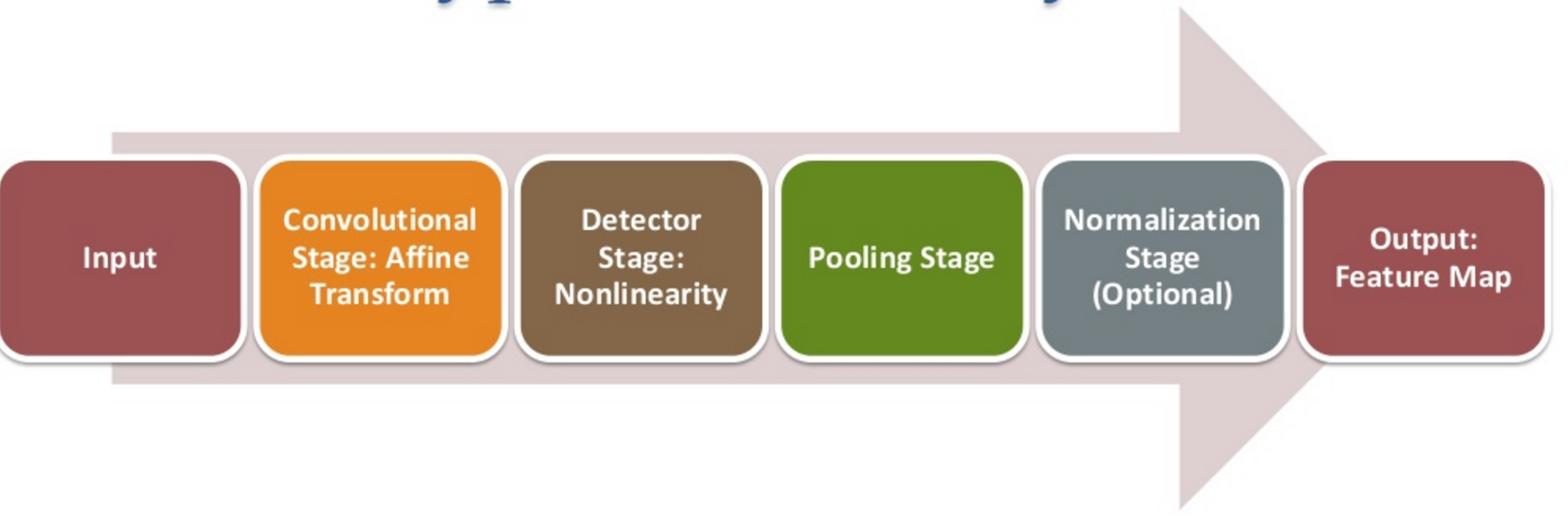
A Convolutional Neural Network (CNN) is comprised of one or more convolutional layers (often with a subsampling step) and then followed by one or more fully connected layers as in a standard [multilayer neural network](http://ufldl.stanford.edu/tutorial/supervised/MultiLayerNeuralNetworks). The architecture of a CNN is designed to take advantage of the 2D structure of an input image (or other 2D input such as a speech signal). This is achieved with local connections and tied weights followed by some form of pooling which results in translation invariant features. Another benefit of CNNs is that they are easier to train and have many fewer parameters than fully connected networks with the same number of hidden units. In this article we will discuss the architecture of a CNN and the back propagation algorithm to compute the gradient with respect to the parameters of the model in order to use gradient based optimization. See the respective tutorials on [convolution](http://ufldl.stanford.edu/tutorial/supervised/FeatureExtractionUsingConvolution) and [pooling](http://ufldl.stanford.edu/tutorial/supervised/Pooling) for more details on those specific operations.

**2.1 Typical CNN layer:**

A CNN consists of several convolutional and subsampling layers optionally followed by fully connected layers. The input to a convolutional layer is a m x m x rm x m x r image where mm is the height and width of the image and rr is the number of channels, e.g. an RGB image has r=3r=3. The convolutional layer will have kk filters (or kernels) of size n x n x qn x n x q where nn is smaller than the dimension of the image and qq can either be the same as the number of channels rr or smaller and may vary for each kernel. The size of the filters gives rise to the locally connected structure which are each convolved with the image to produce kk feature maps of size m−n+1m−n+1. Each map is then subsampled typically with mean or max pooling over p x pp x p contiguous regions where p ranges between 2 for small images (e.g. MNIST) and is usually not more than 5 for larger inputs. Either before or after the subsampling layer an additive bias and sigmoidal nonlinearity is applied to each feature map. The figure below illustrates a full layer in a CNN consisting of convolutional and subsampling sublayers. Units of the same color have tied weights.

**2.2 Structure:**

The architecture of a typical CNN consists of layers of different operations, including convolutional layers, pooling layers and fully-connected layers. Here we explain the CNN structure with 2D image arrays as data input; there are adaptations to work with other forms of data.

****

**2.2.1 Back Propagation**

Let δ(l+1)δ(l+1) be the error term for the (l+1)(l+1)-st layer in the network with a cost function J(W,b;x,y)J(W,b;x,y)where (W,b)(W,b) are the parameters and (x,y)(x,y) are the training data and label pairs. If the ll-th layer is densely connected to the (l+1)(l+1)-st layer, then the error for the ll-th layer is computed as

δ(l)=((W(l))Tδ(l+1))∙f′(z(l))δ(l)=((W(l))Tδ(l+1))∙f′(z(l))

and the gradients are

∇W(l)J(W,b;x,y)∇b(l)J(W,b;x,y)=δ(l+1)(a(l))T,=δ(l+1)

.∇W(l)J(W,b;x,y)=δ(l+1)(a(l))T,∇b(l)J(W,b;x,y)=δ(l+1).

If the ll-th layer is a convolutional and subsampling layer then the error is propagated through as

δ(l)k=upsample((W(l)k)Tδ(l+1)k)∙f′(z(l)k)δk(l)=upsample((Wk(l))Tδk(l+1))∙f′(zk(l))

Where kk indexes the filter number and f′(z(l)k)f′(zk(l)) is the derivative of the activation function. The upsample operation has to propagate the error through the pooling layer by calculating the error w.r.t to each unit incoming to the pooling layer. For example, if we have mean pooling then upsample simply uniformly distributes the error for a single pooling unit among the units which feed into it in the previous layer. In max pooling the unit which was chosen as the max receives all the error since very small changes in input would perturb the result only through that unit.

Finally, to calculate the gradient w.r.t to the filter maps, we rely on the border handling convolution operation again and flip the error matrix δ(l)kδk(l) the same way we flip the filters in the [convolutional layer](http://ufldl.stanford.edu/tutorial/supervised/FeatureExtractionUsingConvolution).

∇W(l)kJ(W,b;x,y)∇b(l)kJ(W,b;x,y)=∑i=1m(a(l)i)∗rot90(δ(l+1)k,2),

=∑a,b(δ(l+1)k)a,b.∇Wk(l)J(W,b;x,y)=∑i=1m(ai(l))∗rot90(δk(l+1),2),∇bk(l)J(W,b;x,y)=∑a,b(δk(l+1))a,b.

Where a(l)a(l) is the input to the ll-th layer, and a(1)a(1) is the input image. The operation (a(l)i)∗δ(l+1)k(ai(l))∗δk(l+1) is the “valid” convolution between ii-th input in the ll-th layer and the error w.r.t. the kk-th filter.

**2.2.2 Convolutional Layer**. A convolutional layer takes the feature maps from the previous layer (the original image if it is the first layer) and perform convolution with a set of weights called filter banks. More precisely, let x be M × N × K array of M × N pixels and K channels, w ∈ R H×W×K×K0 be K0 filters of size (W, H) with K input channels, b the bias term, the convolution can be expressed as

yk 0 = X k w·,·,kk0 ∗ xk + bk`

where ∗ is a 2D convolution. Convolution takes a local patch into account when computing responses, which is usually highly correlated in terms of image statistics; also, it is translation invariant as the filter window scrolls through the entire image. Therefore, it is effective for identifying distinctive motifs of the images.

It is also worth noting that the multi-channel convolution given by is essentially a weighted sum of the convolutions of all the input channels. Some CNN architectures involve 1 × 1 convolutions, which does not make sense as a conventional 2D convolution, but they act as dimensionality reduction operations by combining the input channels.

**2.2.3 Nonlinear Activation.** The local weighted sum acquired from a convolutional layer is passed through a non-linearity that is applied element-wise, e.g. Rectifier Linear Unit (ReLU)

yijk = max {0, xijk}

ReLU performed better in neural networks than sigmoid or hyperbolic tangent because it was free of the gradient saturation problem and preserved sparsity of the signal.

**2.2.4 Pooling Layer.** Pooling operations combine a local patch into one output value as a means of down sampling. The common practice is max-pooling that takes the maximum value of a patch from each channel individually, which has better performance than sum-pooling. There are also attempts to get rid of pooling by increasing the stride of the convolutional layers.

**2.2.5 Normalizations and Overfitting Prevention** There are two major numerical difficulties that deep CNN training is facing: speed and overfitting. Since training data is organized in mini batches in stochastic gradient methods, the network training will be seeing data of different distributions all the time, which is said to be having covariate shift and will slow down convergence. Domain adaptation is usually applied to alleviate the effect. In the context of CNN learning, it has been long known that the network training has faster convergence if the inputs are whitened to have zero means and unit variances and decorrelated. However, even if the inputs are whitened, intermediate responses in the middle layers will still experience changes in distributions due to the weight updates in training, which is called internal covariate shift. Earlier models attempted to introduce normalization to the middle layers with local response normalization, but it required manual tuning and did not adapt well to different datasets and was dropped by later models Ioffe and Szegedy proposed batch normalization (BN) to learn the scale and shift parameters along with the network training, so that the optimization is fully aware of and able to accommodate with the internal normalizations. They argued that to preserve the nonlinear expressive power of the activations, the learned transforms should still be able to represent identity so that the normalization could be reverted if desired. Therefore, for each activation x, BN learns a pair of parameters γ, β such that:

x = x − E[x] /Var[x] ,

y = γˆx + β.

To address the overfitting problem, the easiest and most common method is to augment the dataset via label preserving transforms, e.g. random translation, flip, crop and pixel intensity distortion. proposed dropout to randomly drop neurons during training to prevent co-adapting and showed its efficiency and improvement of neural network performance. Wan et al. generalized dropout to DropConnect which randomly selected a subset of weights to zero, enabling finer control over which connection to drop. Bulo` et al. proposed dropout distillation to improve the inference accuracy of a dropout network by finding the best dropout configuration via another stochastic optimization. On the other hand, maxout networks and multi-bias non-linear activation attempted to modify the receptive behavior of the activations to generate more effective responses, thereby tackling the overfitting issues.

**2.2.6 Feature Map:** After training the parameters of a network, we can proceed to extract feature representations. To do so, we must choose an encoder to map the input feature map of each layer to its representation, i.e we must choose the non-linearity to be used after applying the learned filters to all input locations. A straightforward choice is the use of a natural encoding, i.e. whichever encoding is associated to the training procedure. When using EPLS to train networks, the natural encoding is the non-linearity used to compute the output of each layer. However, different training and encoding strategies can be combined. Encodings that lead to sparse representations have proven to be effective in the literature, e.g. soft-threshold encoding is a popular choice, which involves a tunable meta-parameter to control the desired degree of sparsity. Moreover, the use of polarity split has shown to further improve the performance of many experiments. Polarity splitting considers the positive and negative components where Ol is the concatenation of the positive and negative components of the code. Polarity split results in doubling the number of outputs and is usually applied to the output layer of the network. Summarizing, we train deep architectures by means of greedy layer-wise unsupervised pre-training in conjunction with EPLS and choose a feature encoding strategy for each specific problem. Initial parameters are randomly drawn from N (0, 10−8). Each layer is trained for a minimum of 20 epochs and a maximum of Nl h epochs. If the relative training error decrease between epochs is very small, the training stops. The mini-batch size is initialized to N Nl h and, as is standard practice, the mini-batch size is doubled every time the training error between two consecutive epochs increases.

**3 Regularization:**

For regularization, we employ dropout on the penultimate layer with a constraint on l2-norms of the weight vectors Dropout prevents co-adaptation of hidden units by randomly dropping out—i.e., setting to zero—a proportion p of the hidden units during forward backpropagation. That is, given the penultimate layer

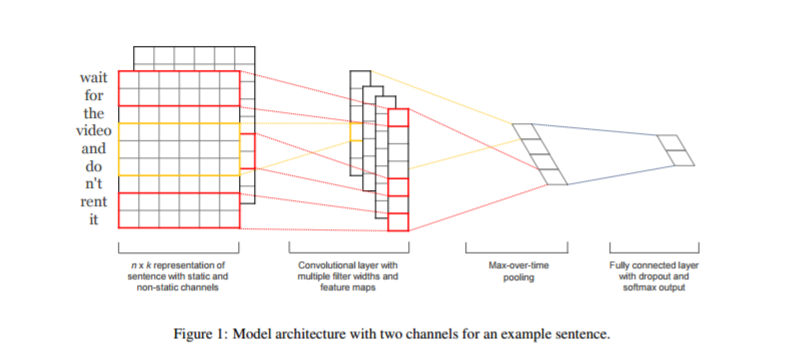
z = [ˆc1, . . . , cˆm] y = w · z + b (4)

for output unit y in forward propagation, dropout uses

y = w · (z ◦ r) + b,

where ◦ is the element-wise multiplication operator and r ∈ R m is a ‘masking’ vector of Bernoulli random variables with probability p of being 1. Gradients are backpropagated only through the unmasked units.

**4 Model:**

The model architecture, shown in figure 1, is a slight variant of the CNN architecture of Collobert. Let xi ∈ R k be the k-dimensional word vector corresponding to the i-th word in the sentence. A sentence of length n (padded where 

necessary) is represented as x1:n = x1 ⊕ x2 ⊕ . . . ⊕ xn, where ⊕ is the concatenation operator. In general, let xi:i+j refer to the concatenation of words xi , xi+1, . . . , xi+j . A convolution operation involves a filter w ∈ R hk, which is applied to a window of h words to produce a new feature. For example, a feature ci is generated from a window of words xi:i+h−1 by ci = f(w · xi:i+h−1 + b).

Here b ∈ R is a bias term and f is a non-linear function such as the hyperbolic tangent. This filter is applied to each possible window of words in the sentence {x1:h, x2:h+1, . . . , xn−h+1:n} to produce a feature map c = [c1, c2, . . . , cn−h+1], with c ∈ R n−h+1. We then apply a max-overtime pooling operation (Collobert et al., 2011) over the feature map and take the maximum value cˆ = max{c} as the feature corresponding to this filter. The idea is to capture the most important feature—one with the highest value—for each feature map. This pooling scheme naturally deals with variable sentence lengths. We have described the process by which one feature is extracted from one filter. The model uses multiple filters (with varying window sizes) to obtain multiple features. These features form the penultimate layer and are passed to a fully connected softmax layer whose output is the probability distribution over labels. In one of the model variants, we experiment with having two ‘channels’ of word vectors—one that is kept static throughout training and one that is fine-tuned via backpropagation In the multichannel architecture each filter is applied to both channels and the results are added to calculate ci in equation. The model is otherwise equivalent to the single channel architecture.

**5 Architecture:**

* Pad input sentences so that they are of the same length.
* Map words in the padded sentence using word embedding (which may be either initialized as zero vectors or initialized as word2vec embedding’s) to obtain a matrix corresponding to the sentence.
* Apply convolution layer with multiple filter widths and feature maps.
* Apply max-over-time pooling operation over the feature map.
* Concatenate the pooling results from different layers and feed to a fully-connected layer with softmax activation.
* Softmax outputs probabilistic distribution over the labels.
* Use dropout for regularization.

**6 Datasets and Experimental Setup**

We test our model on MR movie dataset

* **MR**: Movie reviews with one sentence per review. Classification involves detecting positive/negative reviews.

**6.1 Hyperparameters and Training**

For datasets, we use: rectified linear units, filter windows (h) of 3, 4, 5 with 100 feature maps each, dropout rate (p) of 0.5, l2 constraint (s) of 3, and mini-batch size of 50. These values were chosen via a grid search on the SST-2 dev set. We do not otherwise perform any dataset specific tuning other than early stopping on dev sets. For datasets without a standard dev set we randomly select 10% of the training data as the dev set. Training is done through stochastic gradient descent over shuffled mini-batches with the Adadelta update rule.

**6.2 Pre-trained Word Vectors**

Initializing word vectors with those obtained from an unsupervised neural language model is a popular method to improve performance in the absence of a large supervised training set. We use the publicly available word2vec vectors that were trained on 100 billion words from Google News. The vectors have dimensionality of 300 and were trained using the continuous bag-of-words architecture. Words not present in the set of pre-trained words are initialized randomly.

**6.3 Data Preprocessing**

To process the raw data, run python process\_data.py path where path points to the word2vec binary file (i.e. GoogleNews-vectors-negative300.bin file). This will create a pickle object called mr.p in the same folder, which contains the dataset in the right format. Here is what we see when we run the model. The code and the output is given below for reference:

def clean\_str\_sst(string):

"""

Tokenization/string cleaning for the SST dataset

"""

string = re.sub(r"[^A-Za-z0-9(),!?\'\`]", " ", string)

string = re.sub(r"\s{2,}", " ", string)

return string.strip().lower()

if \_\_name\_\_=="\_\_main\_\_":

w2v\_file = sys.argv[1]

data\_folder = ["rt-polarity.pos","rt-polarity.neg"]

print "loading data...",

revs, vocab = build\_data\_cv(data\_folder, cv=10, clean\_string=True)

max\_l = np.max(pd.DataFrame(revs)["num\_words"])

print "data loaded!"

print "number of sentences: " + str(len(revs))

print "vocab size: " + str(len(vocab))

print "max sentence length: " + str(max\_l)

print "loading word2vec vectors...",

w2v = load\_bin\_vec(w2v\_file, vocab)

print "word2vec loaded!"

print "num words already in word2vec: " + str(len(w2v))

add\_unknown\_words(w2v, vocab)

W, word\_idx\_map = get\_W(w2v)

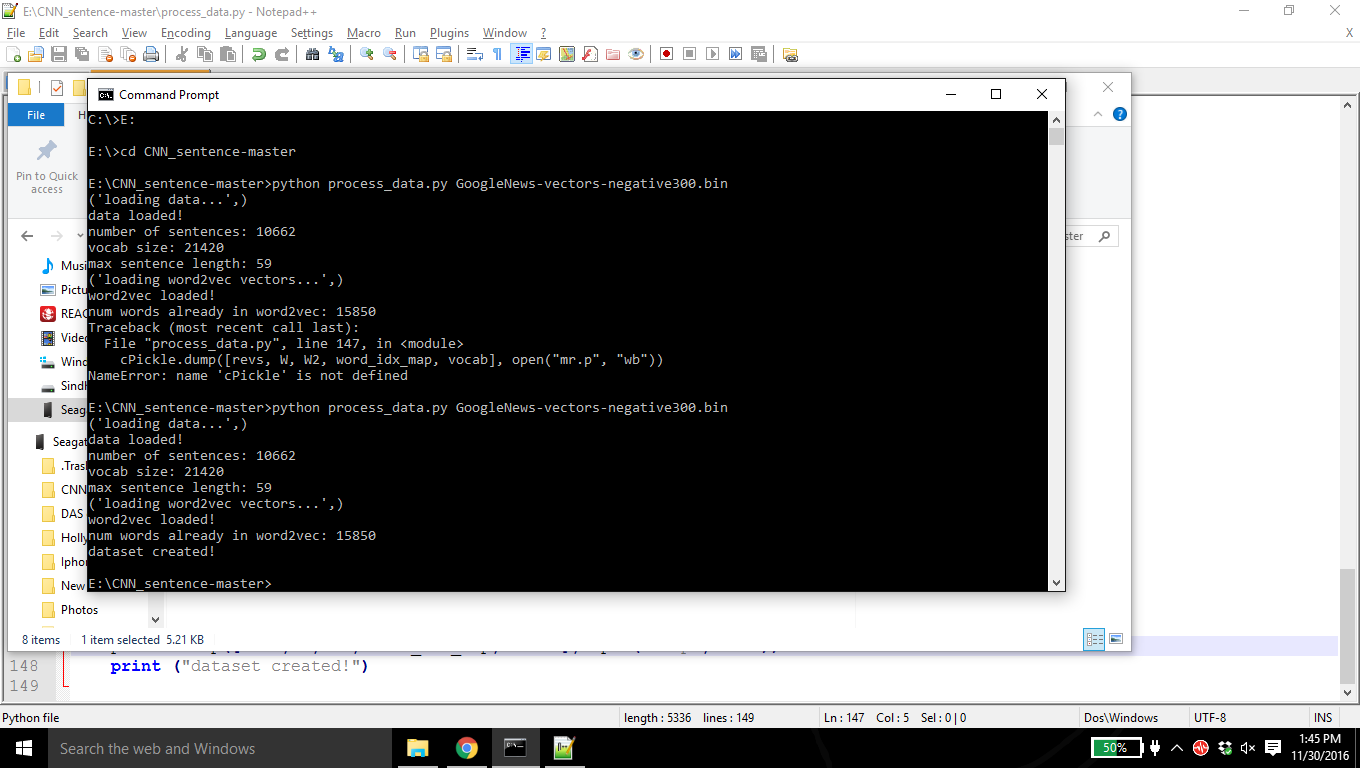
rand\_vecs = {}

add\_unknown\_words(rand\_vecs, vocab)

W2, \_ = get\_W(rand\_vecs)

cPickle.dump([revs, W, W2, word\_idx\_map, vocab], open("mr.p", "wb"))

print "dataset created!"



**6.3.1 Training the dataset code snippet:**

def train\_conv\_net(datasets,

U,

img\_w=300,

filter\_hs=[3,4,5],

hidden\_units=[100,2],

dropout\_rate=[0.5],

shuffle\_batch=True,

n\_epochs=25,

batch\_size=50,

lr\_decay = 0.95,

conv\_non\_linear="relu",

activations=[Iden],

sqr\_norm\_lim=9,

non\_static=True):

"""

Train a simple conv net

img\_h = sentence length (padded where necessary)

img\_w = word vector length (300 for word2vec)

filter\_hs = filter window sizes

hidden\_units = [x,y] x is the number of feature maps (per filter window), and y is the penultimate layer

sqr\_norm\_lim = s^2 in the paper

lr\_decay = adadelta decay parameter

"""

rng = np.random.RandomState(3435)

img\_h = len(datasets[0][0])-1

filter\_w = img\_w

feature\_maps = hidden\_units[0]

filter\_shapes = []

pool\_sizes = []

for filter\_h in filter\_hs:

filter\_shapes.append((feature\_maps, 1, filter\_h, filter\_w))

pool\_sizes.append((img\_h-filter\_h+1, img\_w-filter\_w+1))

parameters = [("image shape",img\_h,img\_w),("filter shape",filter\_shapes), ("hidden\_units",hidden\_units),

("dropout", dropout\_rate), ("batch\_size",batch\_size),("non\_static", non\_static),

("learn\_decay",lr\_decay), ("conv\_non\_linear", conv\_non\_linear), ("non\_static", non\_static)

,("sqr\_norm\_lim",sqr\_norm\_lim),("shuffle\_batch",shuffle\_batch)]

print parameters



**6.3.2 Defining the Model architecture code snippet:**

#define model architecture

index = T.lscalar()

x = T.matrix('x')

y = T.ivector('y')

Words = theano.shared(value = U, name = "Words")

zero\_vec\_tensor = T.vector()

zero\_vec = np.zeros(img\_w)

set\_zero = theano.function([zero\_vec\_tensor], updates=[(Words, T.set\_subtensor(Words[0,:], zero\_vec\_tensor))], allow\_input\_downcast=True)

layer0\_input = Words[T.cast(x.flatten(),dtype="int32")].reshape((x.shape[0],1,x.shape[1],Words.shape[1]))

conv\_layers = []

layer1\_inputs = []

for i in xrange(len(filter\_hs)):

filter\_shape = filter\_shapes[i]

pool\_size = pool\_sizes[i]

conv\_layer = LeNetConvPoolLayer(rng, input=layer0\_input,image\_shape=(batch\_size, 1, img\_h, img\_w),

filter\_shape=filter\_shape, poolsize=pool\_size, non\_linear=conv\_non\_linear)

layer1\_input = conv\_layer.output.flatten(2)

conv\_layers.append(conv\_layer)

layer1\_inputs.append(layer1\_input)

layer1\_input = T.concatenate(layer1\_inputs,1)

hidden\_units[0] = feature\_maps\*len(filter\_hs)

classifier = MLPDropout(rng, input=layer1\_input, layer\_sizes=hidden\_units, activations=activations, dropout\_rates=dropout\_rate)

**6.3.3 Start training over small batches:**

#start training over mini-batches

print '... training'

epoch = 0

best\_val\_perf = 0

val\_perf = 0

test\_perf = 0

cost\_epoch = 0

while (epoch < n\_epochs):

start\_time = time.time()

epoch = epoch + 1

if shuffle\_batch:

for minibatch\_index in np.random.permutation(range(n\_train\_batches)):

cost\_epoch = train\_model(minibatch\_index)

set\_zero(zero\_vec)

else:

for minibatch\_index in xrange(n\_train\_batches):

cost\_epoch = train\_model(minibatch\_index)

set\_zero(zero\_vec)

train\_losses = [test\_model(i) for i in xrange(n\_train\_batches)]

train\_perf = 1 - np.mean(train\_losses)

val\_losses = [val\_model(i) for i in xrange(n\_val\_batches)]

val\_perf = 1- np.mean(val\_losses)

print('epoch: %i, training time: %.2f secs, train perf: %.2f %%, val perf: %.2f %%' % (epoch, time.time()-start\_time, train\_perf \* 100., val\_perf\*100.))

if val\_perf >= best\_val\_perf:

best\_val\_perf = val\_perf

test\_loss = test\_model\_all(test\_set\_x,test\_set\_y)

test\_perf = 1- test\_loss

return test\_perf

**6.3.4 Printing results:**

if \_\_name\_\_=="\_\_main\_\_":

print "loading data...",

x = cPickle.load(open("mr.p","rb"))

revs, W, W2, word\_idx\_map, vocab = x[0], x[1], x[2], x[3], x[4]

print "data loaded!"

mode= sys.argv[1]

word\_vectors = sys.argv[2]

if mode=="-nonstatic":

print "model architecture: CNN-non-static"

non\_static=True

elif mode=="-static":

print "model architecture: CNN-static"

non\_static=False

execfile("conv\_net\_classes.py")

if word\_vectors=="-rand":

print "using: random vectors"

U = W2

elif word\_vectors=="-word2vec":

print "using: word2vec vectors"

U = W

results = []

r = range(0,10)

for i in r:

datasets = make\_idx\_data\_cv(revs, word\_idx\_map, i, max\_l=56,k=300, filter\_h=5)

perf = train\_conv\_net(datasets,

U,

lr\_decay=0.95,

filter\_hs=[3,4,5],

conv\_non\_linear="relu",

hidden\_units=[100,2],

shuffle\_batch=True,

n\_epochs=25,

sqr\_norm\_lim=9,

non\_static=non\_static,

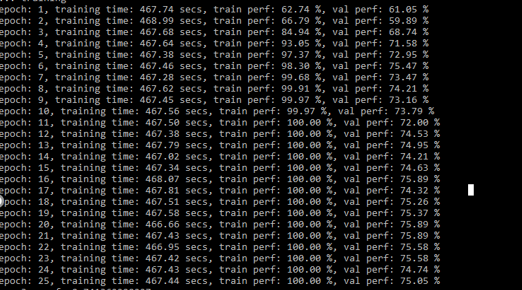
batch\_size=50,

dropout\_rate=[0.5])

print "cv: " + str(i) + ", perf: " + str(perf)

results.append(perf)

print str(np.mean(results))



**Commands for Running the Model:**

THEANO\_FLAGS=mode=FAST\_RUN,device=cpu,floatX=float32 python conv\_net\_sentence.py -nonstatic -rand

THEANO\_FLAGS=mode=FAST\_RUN,device=cpu,floatX=float32 python conv\_net\_sentence.py -static -word2vec

THEANO\_FLAGS=mode=FAST\_RUN,device=cpu,floatX=float32 python conv\_net\_sentence.py -nonstatic -word2vec

**7 Model Variations**

We experiment with several variants of the model.

**• CNN-rand:** Our baseline model where all words are randomly initialized and then modified during training.

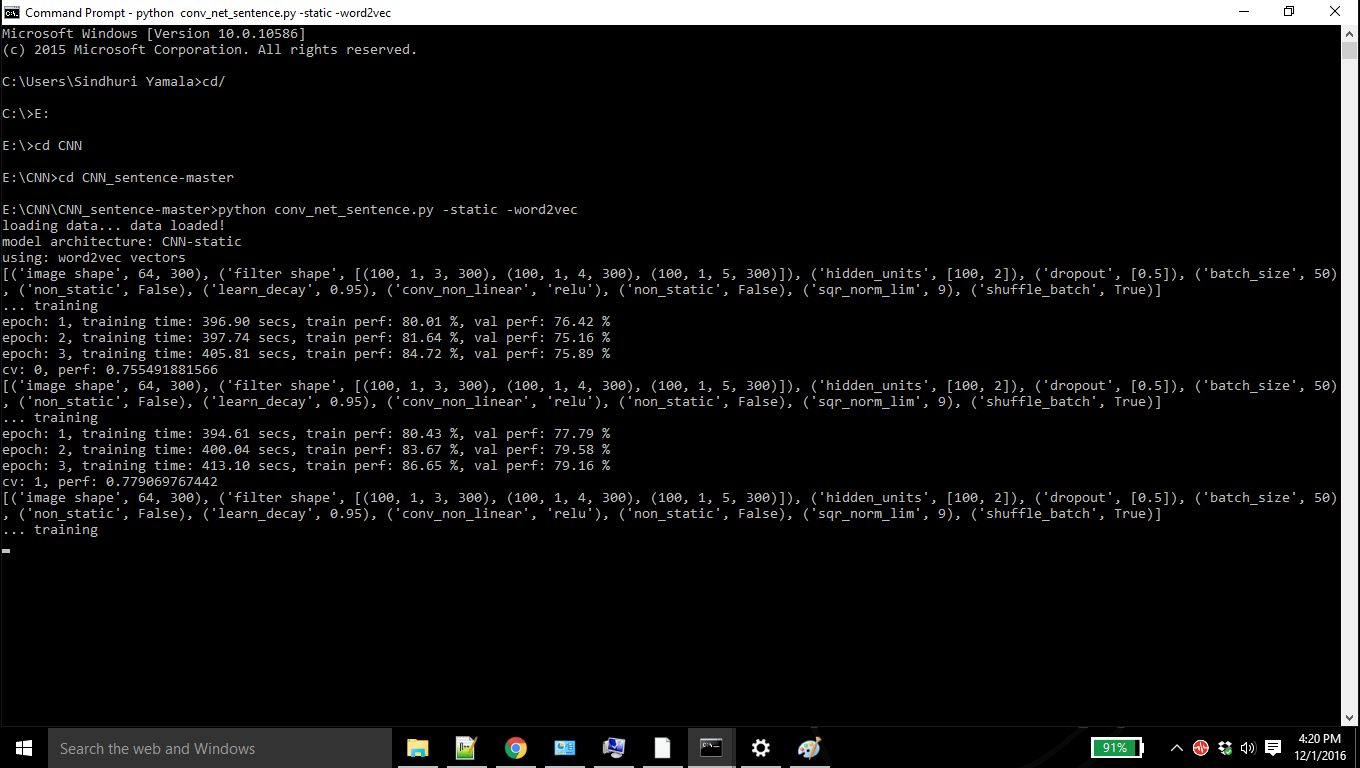
**• CNN-static:** A model with pre-trained vectors from word2vec. All words— including the unknown ones that are randomly initialized—are kept static and only the other parameters of the model are learned.

**• CNN-non-static:** Same as above but the pretrained vectors are fine-tuned for each task.

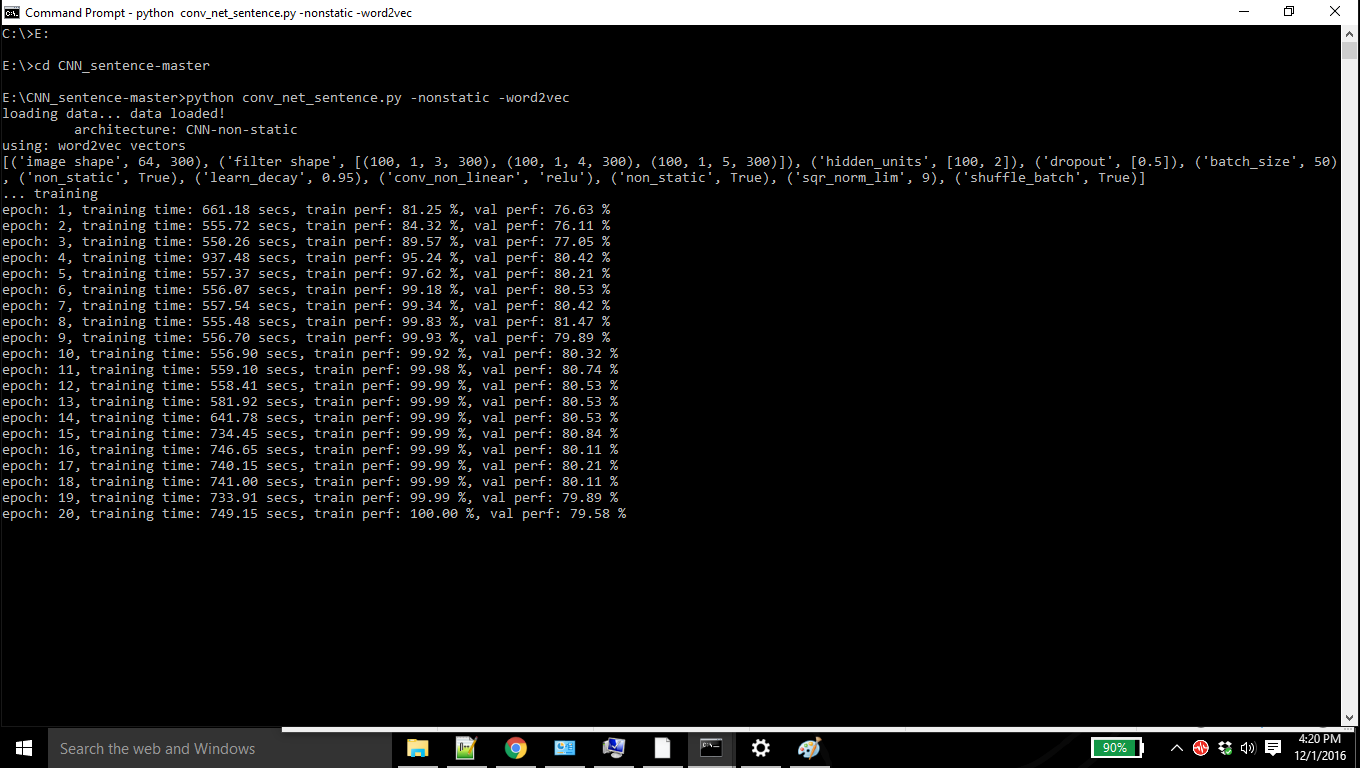
**8 Results:**

Results of our models against other methods are Our baseline model with all randomly initialized words (CNN-rand) does not perform well on its own. While we had expected performance, gains using pre-trained vectors, we were surprised at the magnitude of the gains. Even a simple model with static vectors (CNN-static) performs remarkably well, giving competitive results against the more sophisticated deep learning models that utilize complex pooling schemes. These results suggest that the pretrained vectors are good, ‘universal’ feature extractors and can be utilized across datasets. Finetuning the pre-trained vectors for each task gives still further improvements (CNN-non-static).

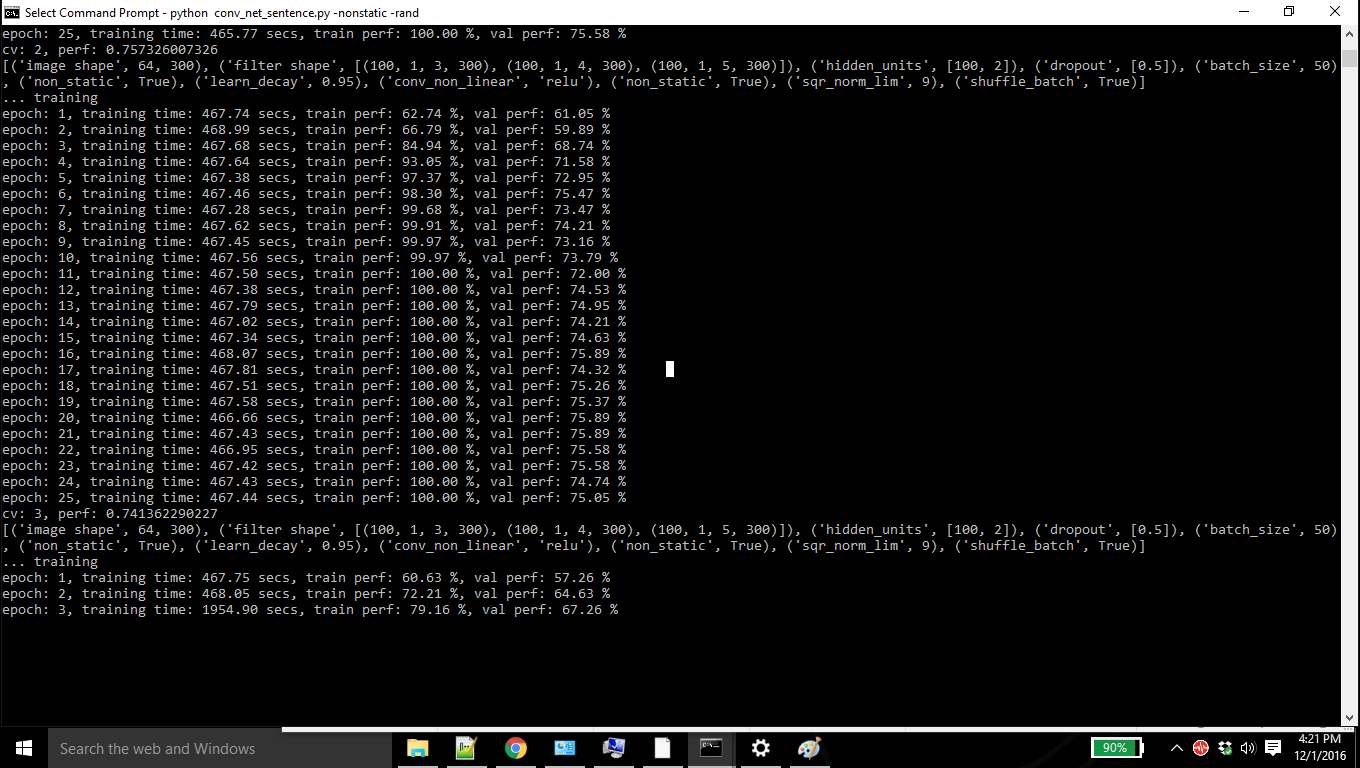
The output for CNN-static



The output for CNN- nonstatic



The output for CNN-Rand



**Static vs. Non-static Representations:**

For (randomly initialized) tokens not in the set of pre-trained vectors, fine-tuning allows them to learn more meaningful representations: the network learns that exclamation marks are associated with effusive expressions and that commas are conjunctive.

As is the case with the single channel non-static model, the multichannel model can fine-tune the non-static channel to make it more specific to the task-at-hand.

**9 Conclusion:**

In the present work, we have described a series of experiments with convolutional neural networks built on top of word2vec. Despite little tuning of hyperparameters, a simple CNN with one layer of convolution performs remarkably well. Our results add to the well-established evidence that unsupervised pre-training of word vectors is an important ingredient in deep learning for NLP.

**Acknowledgments**

We would like to thank Yann LeCun and the anonymous authors for their help.

**References:**

<https://github.com/yoonkim/CNN_sentence>

<http://www.aclweb.org/anthology/D14-1181>

<http://jurix.nl/pdf/j04-05.pdf>

<https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf>

<http://www3.cs.stonybrook.edu/~ziguan/docs/RPEreport.pdf>

<http://cs231n.github.io/convolutional-networks/>

<https://en.wikipedia.org/wiki/Convolutional_neural_network>

<https://ujjwalkarn.me/2016/08/11/intuitive-explanation-convnets/>

<http://cs231n.stanford.edu/>

<http://ufldl.stanford.edu/tutorial/supervised/ConvolutionalNeuralNetwork/>