Homework 4

Benjamin Sorenson, Rahul Bhojwani

December 14, 2017

1. (a) The `AdaGrad` algorithm works much like the standard stochastic gradient descent algorithm, but rather than a fixed global learning rate (ϵ) the learning rate is adjusted at each iteration (t) for each component of $\boldsymbol{\theta}$ by the following two-stage update

$$\mathbf{r}^{(t+1)} = \mathbf{r}^{(t)} + \mathbf{g}^{(t+1)} \odot \mathbf{g}^{(t+1)} \tag{1}$$

$$\Delta\boldsymbol{\theta} = \frac{\epsilon}{\delta + \sqrt{\mathbf{r}^{(t+1)}}} \odot \mathbf{g}^{(t+1)} \tag{2}$$

$$\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} + \Delta\boldsymbol{\theta} \tag{3}$$

Where $\mathbf{r}^{(0)} = 0$, $\mathbf{g}^{(t)}$ is computed as in stochastic gradient descent, and δ is a small constant included for numerical stability. This has the effect of performing larger updates for infrequent parameters and smaller updates for frequent parameters. Because the learning rate is component-wise adjusted at each iteration, there is no need to manually tune the learning rate as you would in SGD.

It's primary flaw is that the accumulation of the squared gradients in the denominator means that the learning rate will asymptotically approach 0 during the course of optimization so optimization may stall prematurely.

(b) The problem of *exploding* or *vanishing* gradients come from the fact that in NNs the objective function is a nesting of several layers of intermediate objective functions taking the form as seen below.

$$f(x) = f_k(f_{k-1}(\dots f_1(x)\dots)) \tag{4}$$

And from the chain rule, the gradient then takes the form

$$f' = f'_k f'_{k-1} \cdots f'_1 \tag{5}$$

This has the effect that even a single layer at an extremum in its gradient can cause the entire gradient to either be very near 0 (vanishing) or very large (exploding) in magnitude. This is more of a problem closer to the output because the gradient of the output is the product of the gradients of all intermediate layers. The gradients of the deeper layers, in contrast, are only the product of the gradients of the layers beneath them.

2. **Regularization Hypothesis:** This hypothesis is that unsupervised pre-training systematically restricts the domain of the parameter space of the cost function to a local basin of

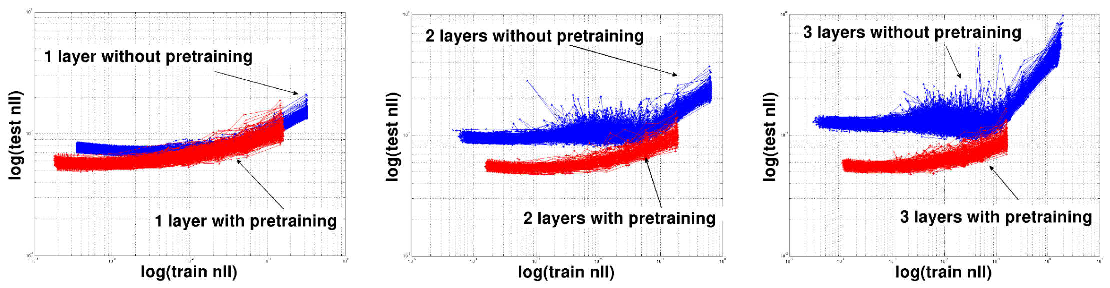

Figure 1: Results from experiment 2

attraction. This is accomplished by learning the dominant features of $\mathbf{X}$ to form a statistically reliable representation of $\mathbf{X}$ (i.e., features of $\mathbf{X}$ that are predictive of the main factors of variation in $P(\mathbf{X})$) at each layer. When learning $P(\mathbf{X})$ is helpful in learning $P(\mathbf{Y}|\mathbf{X})$, this lets us better initialize the training of $P(\mathbf{Y}|\mathbf{X})$. In this sense unsupervised pre-training approach works as an *initialization-as-regularization* strategy, as it acts by defining a particular initialization point for standard supervised training rather than either modifying the supervised objective function or explicitly imposing constraints on the parameters throughout training.

Optimization Hypothesis: Since deep architectures are built of several layers of nonlinearities, they yield an error surface that is non-convex and hard to optimize, with the suspected presence of many local minima. A gradient-based optimization should thus end in the apparent local minimum of whatever basin of attraction we started from. From this perspective, unsupervised pre-training initializes optimization in a region of parameter space where basins of attraction run deeper than when picking starting parameters at random. So, the optimizer is able to converge to a lower point on the error surface.

Common Point: It might also be the case that unsupervised pre-training puts us in a region of parameter space in which training error is not necessarily better than when starting at random (or possibly worse), but which systematically yields better generalization (test error). Such behavior would be indicative of a regularization effect. Note that the two forms of explanation are not necessarily mutually exclusive.

The Evidence: The results from experiments 2 (figure 1) and 3 (figure 2) directly support the regularization hypothesis. Experiment 2 was performed on networks with 1, 2, and 3 layers. As can be seen from figure 1, in networks with more than 1 layer, unsupervised pre-training had a lower test error even though it had a higher training loss. This is evidence that the advantage of unsupervised pre-training is better generalization, not only better optimization. That is, unsupervised pre-training seems to be behaving in the same manner as a good regularizer.

If unsupervised pre-training behaved as a regularizer, we would expect that it's effectiveness would increase with model capacity. As you can see in figure 2, this is exactly what was observed on `MNIST`—as the number of hidden units (model capacity) is increased, the effectiveness of unsupervised pre-training also increases.

The evidence for the optimization hypothesis comes from previous experiments where the

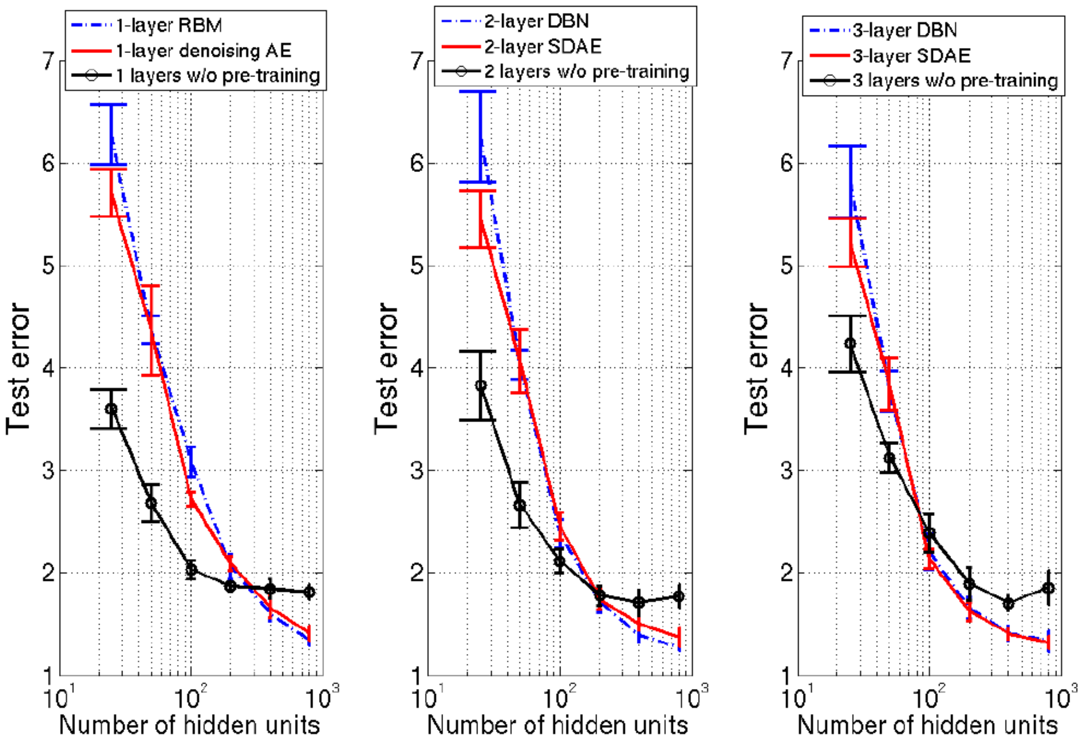

Figure 2: Results from experiment 3

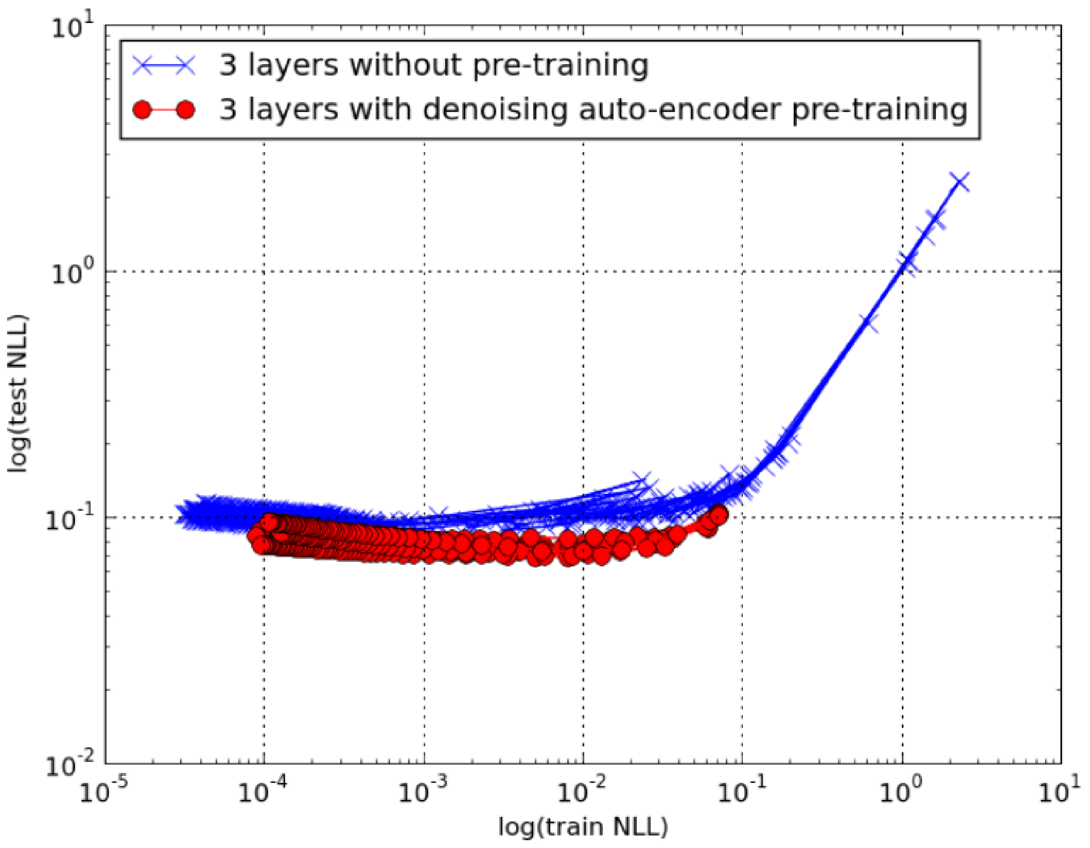

Figure 3: Results from experiment 4

```
with tf.variable_scope('encoder_1') as scope:
    ew1 = tf.get_variable(
        'Weight', [784, 500],
        initializer=tf.truncated_normal_initializer(mean=0.0, stddev=0.1),
        trainable=True)
    eb1 = tf.get_variable(
        'biases', [500],
        initializer=tf.random_normal_initializer(mean=0.0, stddev=0.1),
        trainable=True)
    noise1 = tf.placeholder(tf.float32, [None, 784], name="noise1")
    # Below is the noised data
    C_x = tf.multiply(X, noise1)
    h1_val = tf.add(tf.matmul(C_x, ew1), eb1)
    h1 = tf.nn.relu(h1_val, name=scope.name)
```

Figure 4: First autoencoder layer

top layer of a deep layer was restricted to 20 units, and the training error was compared with and without unsupervised pre-training. The idea was to prevent over-fitting the training error with top layer to make it clear whether some optimization effect was being observed in the lower layers. The previous approach used an early stopping rule (itself a form regularization), and it's conceivable that if this early stopping rule were removed, we may observe different behavior. Figure 3 shows that when the early stopping rule was removed, unsupervised pre-training still achieved lower test error even though its trin error was higher. This again, is evidence in favor of the regularization hypothesis, and against the optimization hypothesis.

3. The model training is as follows

Pretrained model: (a) We have used 3 hidden layer autoencoder. Input is a layer of 784 tensors. We add random noise to it and then pass it to the first encoder where the hidden layer is 500 tensors. After applying nonlinear transformation we decode it again to 784 layered tensor and minimize the cross entropy cost. All three autoencoders follow the same structure—the first is provided in figure 4

$$-(x \log(\hat{x}) + (1 - x) * \log(1 - \hat{x}))$$

(b) We later pass this h1 from encoder 1 with random noise added to our encoder 2. Where is hidden layer is of size 300. We repeat the same steps mentioned above to get h2.

(c) And then we encode h2 with random noise added to get h3 which is a layer of size 150, using the same steps.

(d) After training this model for 10 epochs we have pretrained our weights. Then we proceed with supervised model by adding a logistic regression layer at the end of h3, which give output of 10 classes. And then applying softmax in the end.

Non Pretrained model: Here we just implement step 4 with the weights randomly initialized.