P¿1[0]

DS-GA.1008 Deep Learning March 7, 2017

Solutions to Assignment 1

Team: Peanut Butter *Due: Tuesday, March 7*

Problem 1 - Backpropagation

1. **Nonlinear Activation Functions**

Solution:

Using the chain rule, we can write $\frac{\partial E}{\partial x_{in}}$ as

$$\frac{\partial E}{\partial x_{in}} = \frac{\partial E}{\partial x_{out}} \cdot \frac{\partial P(y=1|x_{in})}{\partial x_{in}}$$

(a) Sigmoid :

$$\begin{aligned}\frac{\partial E}{\partial x_{in}} &= \frac{\partial E}{\partial x_{out}} \cdot \frac{\partial}{\partial x_{in}} \frac{1}{1+\mathrm{e}^{-x_{in}}}\\ &= \frac{\partial E}{\partial x_{out}} \cdot \frac{\mathrm{e}^{-x_{in}}}{(1+\mathrm{e}^{-x_{in}})^2}\\ &= \frac{\partial E}{\partial x_{out}} \cdot \frac{\mathrm{e}^{x_{in}}}{(1+\mathrm{e}^{x_{in}})^2}\end{aligned}$$

(b) Tanh:

$$\begin{aligned}\frac{\partial E}{\partial x_{in}} &= \frac{\partial E}{\partial x_{out}} \cdot \frac{\partial}{\partial x_{in}} \frac{e^{2x_{in}}-1}{e^{2x_{in}}+1}\\ &= \frac{\partial E}{\partial x_{out}} \frac{2e^{2x_{in}}(e^{2x_{in}}+1) - 2e^{2x_{in}}(e^{2x_{in}}-1)}{(e^{2x_{in}}+1)^2}\\ &= \frac{\partial E}{\partial x_{out}} \frac{4e^{2x_{in}}}{(e^{2x_{in}}+1)^2}\end{aligned}$$

(c) ReLU:

$$\begin{aligned}\frac{\partial E}{\partial x_{in}} &= \frac{\partial E}{\partial x_{out}} \cdot \frac{\partial}{\partial x_{in}} max(0, x_{in}) \\ &= \frac{\partial E}{\partial x_{out}} \cdot \frac{\partial}{\partial x_{in}} \begin{cases} 0, & x \leq 0 \\ x, & x > 0 \end{cases}\end{aligned}$$

At $x = 0$, we work out from definition :

$$\lim_{h \to 0^+} \frac{f(0+h) - f(0)}{h} = 1$$
$$\lim_{h \to 0^-} \frac{f(0+h) - f(0)}{h} = 0$$

Hence, the derivative is not defined at $x = 0$.

$$\frac{\partial E}{\partial x_{in}} = \frac{\partial E}{\partial x_{out}} \begin{cases} 0, & x < 0 \\ 1, & x > 0 \end{cases}$$

□

2. **Softmax**

Solution:
We consider the components of X_{in} to be independent of one another.

Using the product rule, we get:

$$\begin{aligned}\frac{\partial (X_{out})_i}{\partial (X_{in})_j} &= \begin{cases} \frac{\partial}{\partial (X_{in})_i} \frac{\mathrm{e}^{-\beta (X_{in})_i} (1 - \mathrm{e}^{-\beta (X_{in})_i})}{(\sum_k \mathrm{e}^{-\beta (X_{in})_k})^2}, & i = j \\ \\ \frac{\partial}{\partial (X_{in})_i} \frac{-\mathrm{e}^{-\beta (X_{in})_i} \mathrm{e}^{-\beta (X_{in})_j}}{(\sum_k \mathrm{e}^{-\beta (X_{in})_k})^2}, & i \neq j \end{cases} \\ &= \begin{cases} -\beta \frac{\mathrm{e}^{-\beta (X_{in})_i}}{\sum_k \mathrm{e}^{-\beta (X_{in})_k}} + \beta \left(\frac{\mathrm{e}^{-\beta (X_{in})_i}}{\sum_k \mathrm{e}^{-\beta (X_{in})_k}} \right)^2 = \beta (X_{out})_i \left((X_{out})_i - 1 \right) & \text{if } i = j \\ \\ \beta \frac{\mathrm{e}^{-\beta (X_{in})_i} \cdot \mathrm{e}^{-\beta (X_{in})_j}}{\left(\sum_k \mathrm{e}^{-\beta (X_{in})_k} \right)^2} = \beta (X_{out})_i (X_{out})_j & \text{if } i \neq j \end{cases}\end{aligned}$$

□

DS-GA.1008 Deep Learning March 7, 2017

Solutions to Assignment 1

Team: Peanut Butter *Due: Tuesday, March 7*

Problem 2 - Techniques

1. **Optimization**

   Solution:

   Given an objective function $f(\theta)$, The following equations denote the gradient descent weight updates at time step t.

   - **Gradient descent step by momentum method**
     $v_{t+1} = \mu \cdot v_t - \epsilon \delta f(\theta_t)$
     $\theta_{t+1} = \theta_t + v_{t+1}$

   - **Gradient descent step by Nesterov Accelerated descent**
     $v_{t+1} = \mu \cdot v_t - \epsilon \delta f(\theta_t + \mu \cdot v_t)$
     $\theta_{t+1} = \theta_t + v_{t+1}$

   where ϵ is the learning rate, $\mu \in [0, 1]$ is the momentum co-efficient, $\delta f(\theta_t)$ is the gradient at θ_t Momentum method has the effect of speeding up convergence by incorporating a momentum factor in the direction of gradient descent that has persistent reduction. By applying a multiplicative constant between [0,1], this has the effect of pushing down the directions which contribute less and keeping the directions that have persistent change. This accelerates convergence to local minimum requiring fewer iterations than gradient descent without momentum.

   NAG method is very similar to momentum method and differs only in the precise update of the gradients. As we can observe from the weight update, NAG first adds the momentum component to the velocity vector $\mu \cdot v_t$ and then makes a correction if any one of the velocity vector update is a poor choice. In the case of a situation where the update of velocity vector results in an undesirable increase in the objective function , $\delta f(\theta + \mu \cdot v_t)$ would steer the update back to θ_t more strongly than $\epsilon \delta f(\theta_t)$. While the optimization path taken by classical momentum has a lot of fluctuations, NAG avoids a lot of these oscillations and converges smoothly and quickly.

   $\square$

2. **Reducing Overfitting**

   Solution:

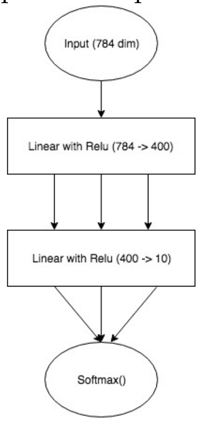
- **Dropout vs. Ensemble**

  - **Dropout**
    Dropout is a technique that can be applied to deterministic feedforward architectures that predict an output y given input vector v. These architectures contain a series of hidden layers h = $\{h_1, ..., h_L\}$. Dropout trains an ensemble of models consisting of the set of all models that contain a subset of the variables in both v and h.
    To parametrize the variables of the model, a parameter θ is used to define a distribution $p(y|v; \theta, \mu)$, such that, μ is the is a binary mask determining which variables to include in the model.
    On each presentation of a training example, a new sub-model is trained by following the gradient of log $p(y|v; \theta, \mu)$ for a different randomly sampled μ.

  - **Ensemble**
    In ensemble technique, instead of learning a single classifier on a data set, we learn multiple different classifiers. For instance, we might train a decision tree, a perceptron, a KNN, and multiple neural networks with different architectures. Call these classifiers $\{f_1, ..., f_M\}$. At test time, a prediction is made by by voting. Another technique called bagging is used where many different models are trained on different subsets of the data.

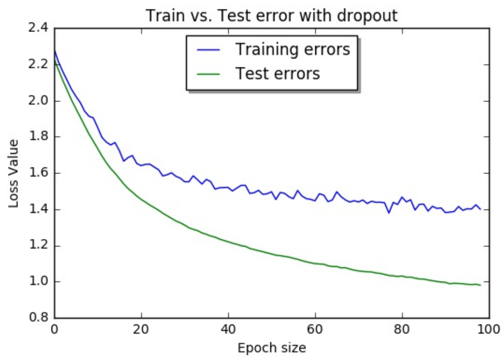
  - **Differences**
    Dropout training differs from bagging and ensemble in that each model is trained for only one step and all of the models share parameters. For this training procedure to behave as if it is training an ensemble rather than a single model, each update must have a large effect, so that it makes the sub-model induced by that μ fit the current input v well.

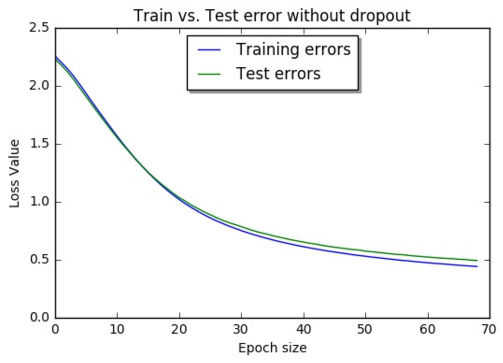
- **Dropout behaviour in Simple Network**
  For the purpose of exploring Dropouts, we used the below simple network structure.

  network.jpg
  We introduced a dropout layer with leave probability of 0.5 and plot the test error's observed with and without droputs.
  To observe the effect of regularization that Dropout has on the model weight parameters,

we compare the norm of the weight vectors in both the models during both testing and training times. The L2 norm of the layer weights remained almost the same after testing and training. During the process of training, Dropout causes some of the network weights to be set to zero (depending upon the leave-propobability). During Testing, Dropout causes the network weights to be scaled by the factor of p.





- **Data Augmentation**

  The common data augmentation methods are, translation, rotation, scaling, contrast variation and colour variations [5]. However, as MNIST dataset has only one colour channel, contrast and colour variations have no effect here.

  (a) **Rotation** The input image shall be rotated (about $\pm 20°$) before passing into the training.

  (b) **Translation** The input image shall be moved across the x and y (or one of them) direction by a small fraction of the input image size (we use $\pm 20\%$), and the empty areas generated shall be filled with the nearest points from the image to avoid sudden colour shifts.

  (c) **Scaling** The input image shall be upscaled or downlscaled randomly by a small factor (we use 0.7x to 1.3x), and then refit to the original image dimensions. If the image is downscaled, then the empty areas shall be filled with reference the nearest pixel to each point.

$\square$

3. **Initialization**

Solution:

- **He vs. Xavier Initializations**

  The solution to a non-convex optimization algorithm (like stochastic gradient descent) depends on the initial values of the parameters.

  - **Xavier Initialization**

    Consider the hypothesis that we are in a linear regime at the initialization, that the weights are initialized independently and that the inputs features variances are the same. From a forward-propagation point of view, to keep information flowing we would like that where z_i is an input feature at ith layer.

$$\forall(i,i'), Var[z_i] = Var[z_{i'}]$$

Similarly, for back-propagation, we want,

$$\forall(i,i'), Var[\frac{\partial Cost}{\partial s_i}] = Var[\frac{\partial Cost}{\partial s_{i'}}]$$

As a compromise to these two conditions, we want to have,

$$\forall i, Var[W_i] = \frac{2}{n_i + n_{i+1}}$$

Where W is a weight matrix where each element was drawn from an IID initialized from a uniform distribution:

$$U[-\frac{1}{\sqrt{n}}, \frac{1}{\sqrt{n}}]$$

and n_i is the width of the layer i. For this to satisfy for every layer, we need the width of each layer to be the same. The standard initialization gives rise to variance with the following property:

$$n * Var[W] = \frac{1}{3}$$

where n is the layer size (assuming all layers of the same size). This will cause the variance of the back-propagated gradient to be dependent on the layer (and decreasing).

– **He Initialization**
Consider a general case, where the initializations may not be linear. For a conv layer, a response is:

$$y_l = W_{lxl} + b_l$$

where W is a d-by-n matrix, where d is the number of filters and each row of W represents the weights of a filter.

$$Var[y_l] = n_l Var[w_l x_l]$$

Assume w_l have zero mean. Then the variance of the product of independent variables gives us:

$$Var[y_l] = n_l Var[w_l] E[{x_2}^2]$$

It can be shown that

$$\frac{1}{2} \cdot n_l Var[w_l] = 1, \forall l$$

This leads to a zero-mean Gaussian distribution whose standard deviation (std) is $\sqrt{\frac{2}{n_l}}$ This can be arrived at from backward propagation as well.

– **Differences**

- * The derivation of Xavier initialization is based on the assumption that the activations are linear. This assumption is invalid for ReLU and PReLU. Hence, He initialization is more general.
  * The Xavier initialization can be implemented as a zero-mean Gaussian distribution whose standard deviation is $\sqrt{\frac{1}{n}}$. At layer L, the std will be $\frac{1}{\sqrt{2^L}}$ that of He initialization.
  * This also causes He initialization to make deeper models start reducing errors faster than Xavier initialization in deeper networks, say with 22 layers. In really deep networks, say, with 30 layers, He initializations are shown to converge but Xavier initializations completely stall.

- **Pre-training** Pre-training is also used for initializing neural networks. This is used since bad initialisation can stall learning due to the instability of gradient in deep nets.
  - **VGG**
    VGG team used random initializations for training shallow models. For deeper models, they train below model with random initializations.

ConvNet Architecture
input (224 * 224 RGB image)
conv3-64
maxpool
conv3-128
maxpool
conv3-256 conv3-256
maxpool
conv3-512 conv3-512
maxpool
conv3-512 conv3-512
maxpool
FC-4096
FC-4096
FC-1000

Table 1: VGG - ConvNet A used for pre-training

In the ConvNet architecture, conv3-128 represents a convolutional network with 3 x 3 kernels and 128 input channels and FC represents a fully connected layer.
While training deeper architectures, they initialised the first four convolutional layers and the last three fully connected layers with the layers of net A and the intermediate layers were initialised randomly. They did not decrease the learning rate for the pre-initialised layers, allowing them to change during learning. For random initialisation (where applicable), they sampled the weights from a normal distribution with the

zero mean and 10^{-2} variance.

- **Unsupervised Feature Learning**

  Coates et. al. used below techniques to learn features for unsupervised training.

  - **Sparse auto-encoder:** An autoencoder with K hidden nodes is trained to minimize squared reconstruction error.The algorithm outputs network weights $W \in R^{M*N}$ and biases $B \in R^{M}$ such that $f(x) = sigmoid(Wx + b)$.
  - **Sparse restricted Boltzmann machine:** The restricted Boltzmann machine (RBM) is an undirected graphical model with K binary hidden variables. Their training is similar to that of auto-encoders in that they also output weights and biases. The training parameters for RBM are decided using cross validation.
  - **K-Means clustering:** K centroids are learnt from the K-means clustering algorithm. Given the learned centroids $c^{(k)}$ from input, 2 feature mapping schemes are learnt -
    - K-means hard

      $$f_k(x) = \begin{cases} 1, & \text{if } k = argmin_j ||c^{(j)} - x||_2^2 \\ 0, & \text{otherwise} \end{cases}$$

    - K-means triangle - this is a non-linear mapping that attempts to be softer than the above encoding

      $$f_k(x) = max\left\{0, \mu(z) z_k\right\}$$

      where $z_k = ||c^{(j)} - x||_2$ and $mu(z)$ is the mean.
  - **Gaussian Mixture Model:** A Gaussian mixture model is a probabilistic model that assumes all the data points are generated from a mixture of a finite number of Gaussian distributions with unknown parameters. A single run of K-means is run to initialize this model.The feature mapping f maps each input to the posterior probabilities.

Since our model was relatively simple, we only used random initializations. It converged with random initializations without any difficulty. We also used batch-norms to accelerate training by avoiding internal covariate shift.

□

DS-GA.1008 Deep Learning March 7, 2017

Solutions to Assignment 1

Team: Peanut Butter *Due: Tuesday, March 7*

Problem 3 - MNIST

We performed experiments on the MNIST dataset to classify images into the digits that they represent. Our final submission consists of a convolutional network, using surrogate classes and on-line labeling to learning from unlabeled data.

We experimented with Pseudo-labels, Surrogate Classes, Variational Auto-Encoders and Ladder Networks and have noted our results below. We applied three types of augmentations (translation, rotation, scaling) to the dataset at runtime to increase variance in the training data, in order to make up for the small amount of labeled data being available. All of our networks used batch normalization and dropout.

Link to models: 99.46% on test set, 99.28% on test set.

- **Model & Loss function**

  The first convolution layer of our model takes the 1x28x28 image and learns 64 5x5 kernels. The second layer further learns 128 5x5 kernels, and the third convolution layer learns 256 5x5 kernels. After each convolution layer, we use ReLU nonlinearity and perform a 2x2 maxpooling and apply batch normalization. We also apply dropouts to each convolution layer, with the dropout probability increasing as the layer goes by. These are then flattened to have 1024 channels. We then have a fully connected layer, which takes in 1024 inputs and sends it over to 512 neurons, and the final layer maps it to 10 output channels, one for each digit.

  The Loss function we use is negative log-likelihood (NLL). We finally have a final logsoftmax layer that outputs the log of the probabilities of each digit, using which, our model predicts the output.

- **Data Augmentation**

  At each epoch of our training, we artificially introduce some distortions to make the network insensitive to in-class variability. These distortions are a combination of translation, rotation, and zooming, with the following parameters randomly chosen:

  1. Rotation of [-30° to 30°]
  2. Translation of [-5 pixels to 5 pixels]
  3. Zooming of [0.7x to 1.3x]

  We also added random Gaussian noise to our images, which seemed to help in providing variance to the dataset and helped improve the validation accuracy.

ConvNet Architecture
input (28 * 28 BW image)
conv5-64
maxpool
batch-norm
dropout(.2)
conv5-128
maxpool
batch-norm
dropout(.3)
conv5-256
maxpool
batch-norm
dropout(.5)
FC-500
FC-10

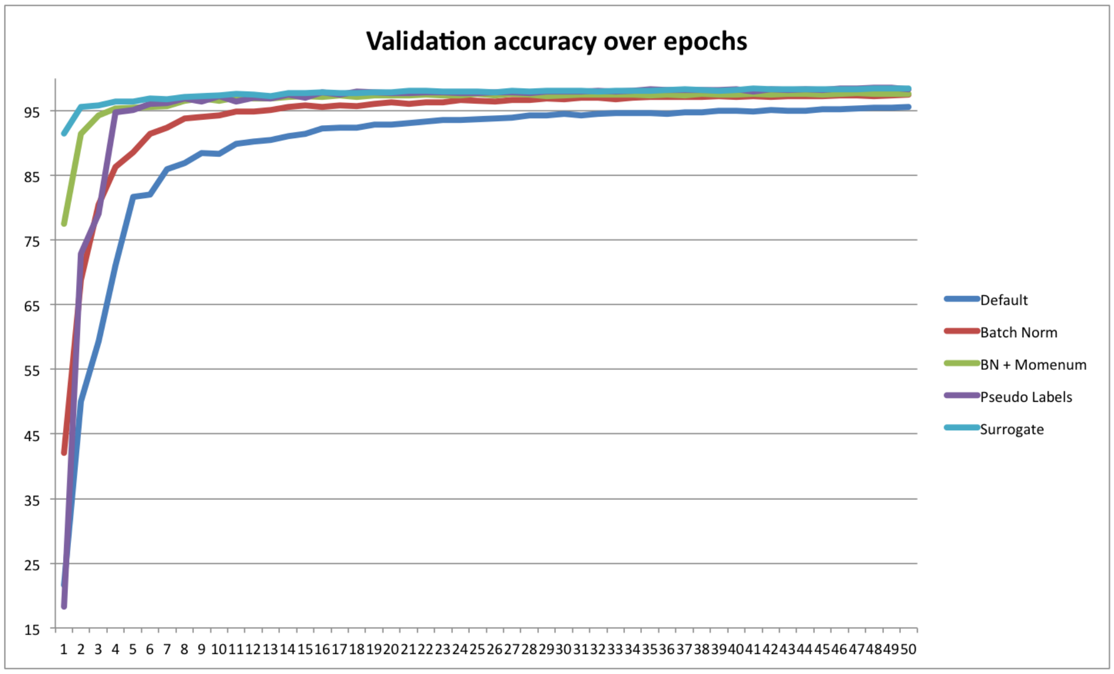
Table 2: Model used in our experiments

- **Training and Regularization**

  We trained all of our models for 300 epochs, with no stopping criterion, while incrementally saving better performing models (determined by validation accuracy).

  For regularizing the model, we initially experimented with adding batch normalization and dropout to our network. This helped the model improve the validation accuracy as the regularization prevented the network from overfitting to the small training set.

  We then added a momentum of 0.8 to improve the convergence speed of our network. The result of these experiments are shown below.



- **Unlabeled images**

  We used several strategies to leverage unsupervised images to improve our performance, which we have described below.

  1. **Surrogate Classes**

     We implement the methods described in [5], and do the following:

     (a) Randomly select 2048 unlabeled images

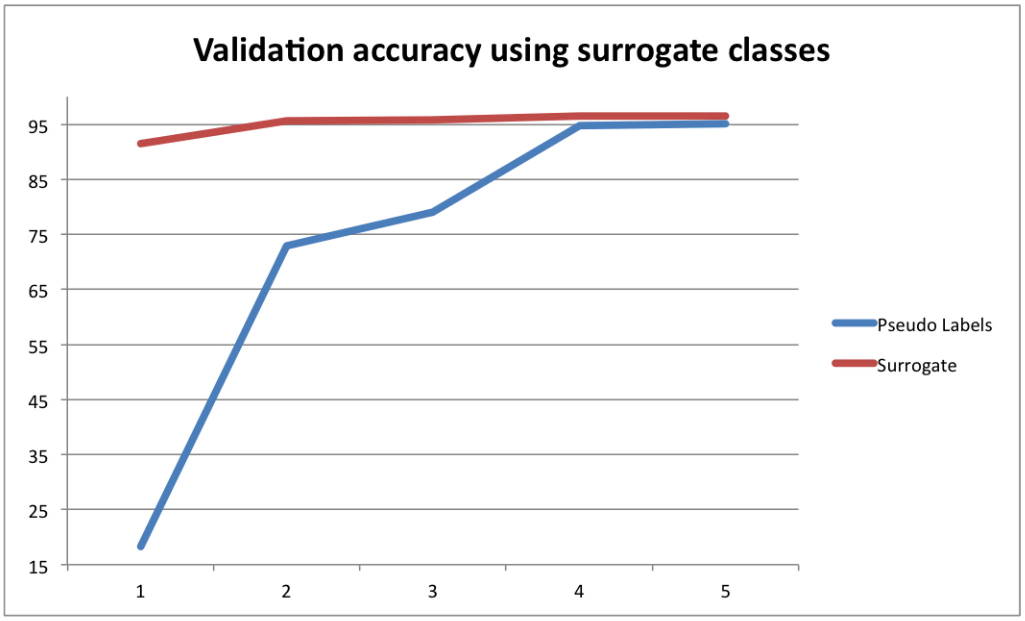
     (b) Generate 16 distortions $\{s_1, s_2, \ldots, s_{16}\}$ to each of them. Let $S = \{s_i, i = 1, \ldots, 16\}$.

     (c) Assign a unique ID, u_i to S_i to serve as the label.

     (d) Train the network such that the output of each image in S_i is expected to be u_i.

     (e) Repeat $1 \ldots 3$.

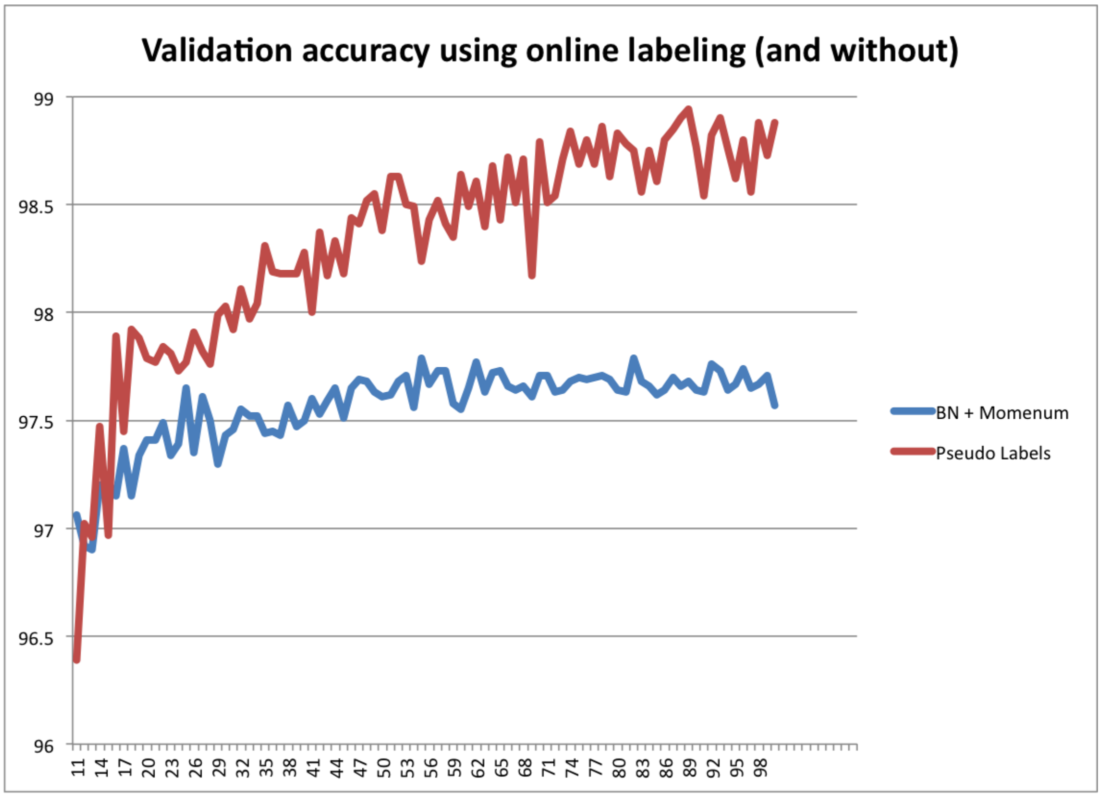
     After several rounds of processing the unlabeled images, we feed in the labeled data. The network, which has already learned useful representations of the images, immediately adapts to predict the output among 10 classes. The speed of convergence can be seen below, when compared to a network that was only trained on labeled images. Even after several epochs, the model trained on surrogate classes proved more accurate than the one that was trained without it (99.29% vs 99.12%).



2. **Pseudo labels**

   In order to use the unlabeled dataset to improve our accuracy, we label it at runtime with our existing model and then train on this newly generated dataset and backprop the loss. At every epoch, the labels of the unlabeled dataset are updated as per the model predictions.

   Labeling the unlabeled dataset at runtime provided over a 1% increase in accuracy as compared to the model that just used batch normalization and momentum, as shown below. This technique is a modified implementation of pseudo-labels. [3]. In pseudo-labels, unlabeled data is used with pseudo labels right from the first epoch, but we used it only after the model stabilized.



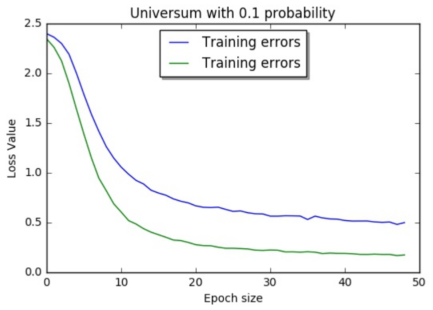
3. **Universum Prescription** Universum Prescription from [1] talks about various ways of labelling unlabelled data, which could later be used in a supervised training model. We specifically explored Dustbin-bin, where-in all the unlabelled data is attributed a dustbin class [usually (n+1) where n is the total number of classes]. The network architecture used for training is the same as any supervised procedure except for the last layer. [1] argues that this has an effect of adding regularization and introduces smoothness in the model. Our best performance came from setting p to . where p is the probability of sampling unlabelled data (called as a regularization parameter. We observed that if the proportion of unlabelled samples (p) exceeds 0.3 , performance starts deteriorating. This is directly correlated with the number of classes.

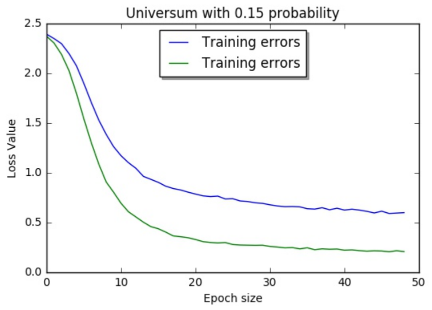
   Optimal choice of probability

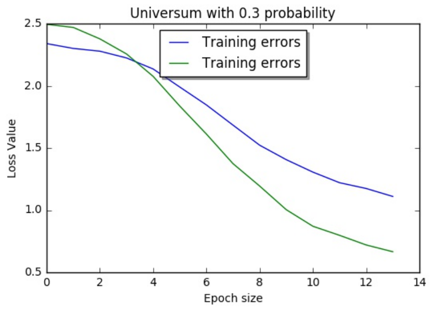
   Say, there are N classes and we have a labelled dataset where the distribution of data across classes is uniform. If T is the total size of the dataset, then we have $\frac{T}{N}$ number of samples belonging to each class. If we introduced an unlabelled dataset, with probability of sampling unlabelled data set at p, then the size of the total dataset is $T + X$, where $X = \frac{p}{1-p}T$ with X number of samples belonging to dustbin class.

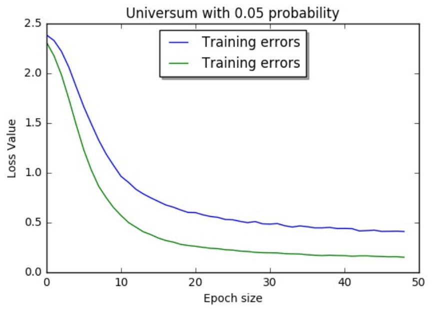
   The number of samples across all classes is now $\{\frac{T}{N}, \frac{T}{N}, ... \frac{p}{1-p} \cdot T\}$ and solving for p, we get:
   p $= \frac{1}{N+1}$

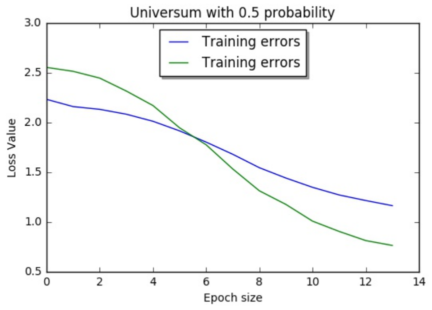
4. **Auto-encoders** To make optimal use of unlabelled dataset, we tried to train a set of generative classifiers where the generative model is essentially an Auto-encoder.


Universum with 0.1 probability
Training errors
Training errors
Loss Value
Epoch size


Universum with 0.15 probability
Training errors
Training errors
Loss Value
Epoch size


Universum with 0.3 probability
Training errors
Training errors
Loss Value
Epoch size


Universum with 0.05 probability
Training errors
Training errors
Loss Value
Epoch size


Universum with 0.5 probability
Training errors
Training errors
Loss Value
Epoch size

There are two parts to a generative classifier :

- Generative model : Network which learns to reconstruct the original input from the internal representation.
- Classifier: Network which makes use of generative's models' parameters to use in the classification task.

We tried couple of different approaches to couple the classifier with the generative model .

Approaches to tie classifiers with Autoencoders

(a) **Decoupled Classifier**: The generative model is trained independently of the classifier. In this case, the out of the generative model would be the reconstructed version of data, which is fed into a supervised classifier. This approach did not perform well for classification. One reason we could think of is, that the latent variables that the generative model learns to reconstruct data is somehow not helpful in classification. [2] explains that *the autoencoder will try to preserve all the details needed for reconstructing the image at pixel level, even though classification is typically invariant to all kinds of transformations which do not preserve pixel values.*

(b) **Coupled Classifier** : To overcome the problem of the above approach, we tried incorporating the classifier inside a generative model. The loss function accordingly would be the sum of
loss = Reconstruction loss + Kl divergence + classification loss

Experimental results

(a) **Variational Auto-encoder** VAE tries to learn the latent variable representation of the data by forcing the latent variables to follow a unit gaussian distribution. Intuitively the network learns the parameters of the data distribution corresponding to each classes. The paramaters learnt are the mean and standard deviation corresponding to each data-distribution

Observations

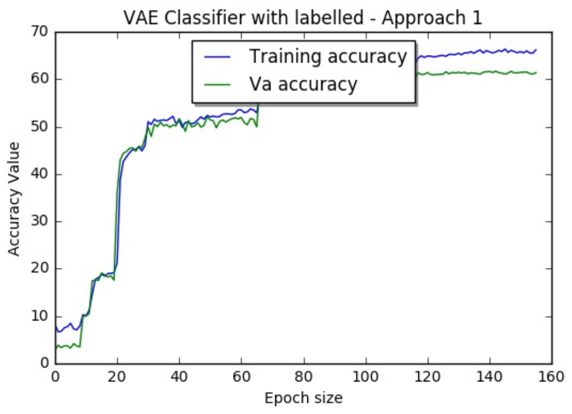
i. Approach 1 : The reconstructed input is fed into a classifier. We ran two experiments, including and excluding unlabelled data. Refer ??

ii. Approach 2 : The size of latent variables is set to the num of classes and SoftMax(latent vars) is treated as the output. Intuitively this should make sense, as the model tries to learn the parameters of the distribution before using the parameters in classification. We experimented with both Softmax(mean) and Softmax(log variance). This approach did not work too.

iii. The maximum val. accuracy we got was closer to 90%.

iv. Since the loss term is a sum of Reconstruction loss + KL divergence + classification term, each of which is of a different scale, the backpropogation optimizes that loss which is higher in magnitude. In our case, recon loss was of the order of (400's etc.) while class loss was (less than 2). The network tries to optimize the recon loss more than the class loss and hence the improvements in classification was quite slow.

(b) **Denoising Auto-encoder** The input to the Auto-encoder is a noised version of the data $\tilde{x}$. The Auto encoder tries to reconstruct the original data x from the



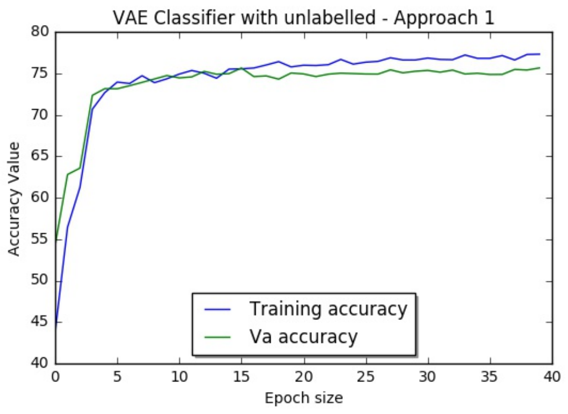
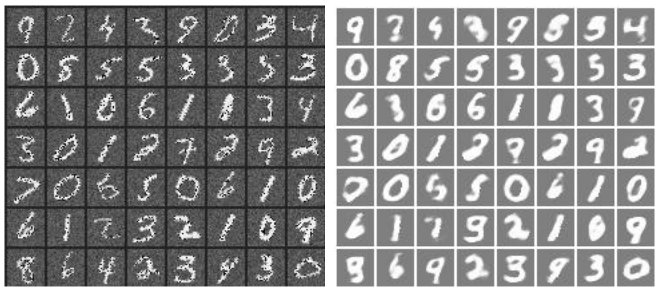


Figure 1: Denoising auto-encoder



noised-data $\tilde{x}$. Although, the denoising-autoencoder performs well in reconstructing the inputs, its classification performance is similar to normal VAE.

(c) **Convolutional Auto-encoder** Adding few convolutional layers into the encoder and decoder part of the AE did not improve the performance.

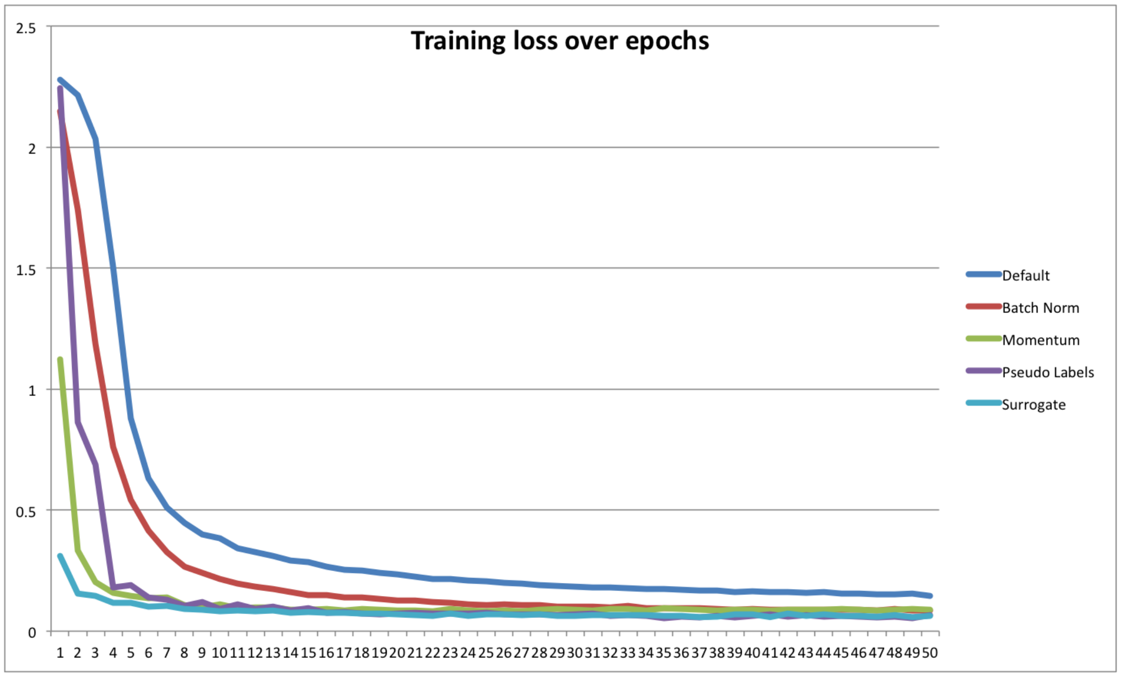
- **Results**

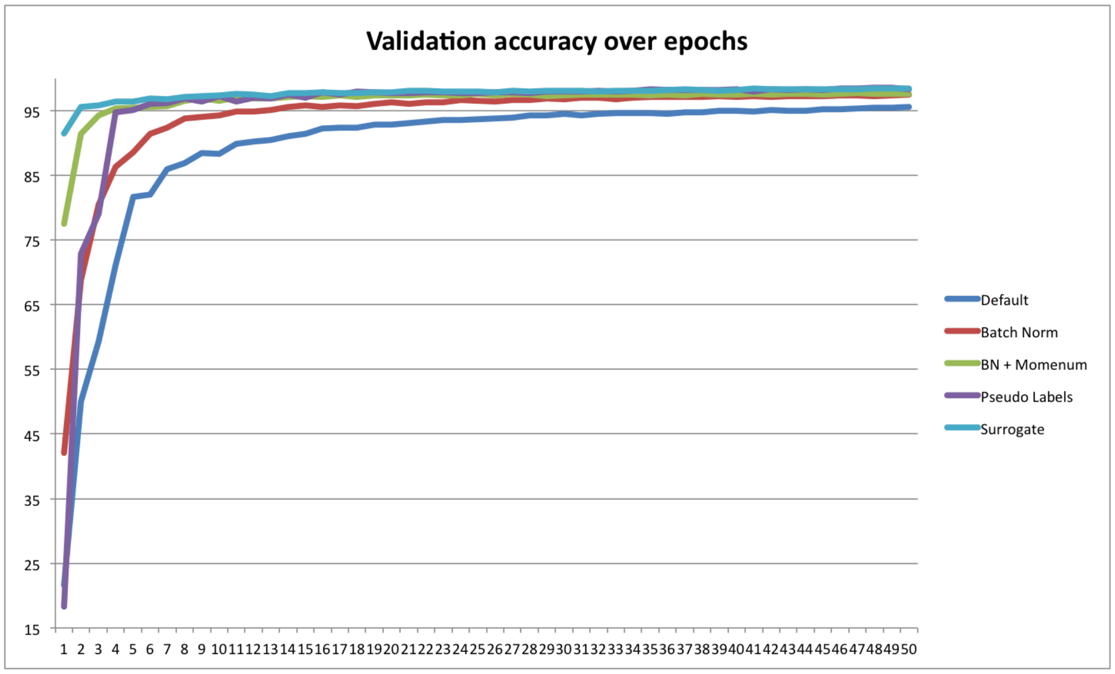
  We made incremental improvements to the default model provided, and have recorded our best validation accuracies for each of them below:

Index	**Network**	**Accuracy**
1	default	0.9520
2	batch normalization	0.9626
3	(2) + pseudo labeling of unlabeled data	0.9754
4	(3) + data augmentation	0.9910
5	(4) + gaussian noise	0.9916
6	(5) + oversampling	0.9928
7	(4) + surrogate classes	**0.9946**

Table 3: Test accuracies with different network structures.

We include the loss and accuracy plots of all our models over the first 50 epochs below.





We experimented with both SGD and Adam for the choice of optimizer, and, while we could not see a big difference in performance of either of them over a large number of epochs, our best performing model used Adam as the optimizer.

References

[1] Universum Prescription: Regularization using Unlabeled Data
https://arxiv.org/abs/1511.03719

[2] Semi-Supervised Learning with Ladder Networks
https://arxiv.org/pdf/1507.02672.pdf

[3] Pseudo-Label: The Simple and Efficient Semi-Supervised Learning Method for Deep Neural Networks
http://deeplearning.net/wp-content/uploads/2013/03/pseudo_label_final.pdf

[4] Maxout Networks
http://jmlr.csail.mit.edu/proceedings/papers/v28/goodfellow13.pdf

[5] Discriminative Unsupervised Feature Learning with Convolutional Neural Networks
http://papers.nips.cc/paper/5548-discriminative-unsupervised-feature-learning-with-convol
pdf