B.Comp Dissertation

Incremental Training of Neural Network with Knowledge Distillation

Author:
David Heryanto

Supervisor:
Prof. Tat-Seng Chua

Department of Computer Science
School of Computing

April 6, 2015

B.Comp Dissertation

Incremental Training of Neural Network with Knowledge Distillation

Author:
David Heryanto

Supervisor:
Prof. Tat-Seng Chua

Project No: H009610
Deliverables: 1. Report
2. Source code
https://github.com/davidheryanto/incremental-kd

Department of Computer Science
School of Computing

April 6, 2015

Abstract

Training a neural network on a huge dataset is challenging because it requires massive computational resources as well as long period of training and test time. On the other hand, simply training a neural network multiple times using different partitions of the dataset is known to deliver poor performance because of issues such as *catastrophic forgetting* (Goodfellow et al., 2013). Other techniques such as ensemble methods (Dietterich, 2000; Džeroski et al., 2009) or mixture of experts (Jacobs et al., 1991) require all trained models to be kept at prediction time, which may severely affect runtime performance and storage size.

In this project, we propose a novel way to combine neural networks trained on different subsets of the dataset. Our approach does not require all the networks to be kept at runtime, producing only a single network that has a comparable generalization performance with respect to the baseline model trained on the whole dataset.

Our preliminary experiments show that we are able to transfer some knowledge from the previous networks to a new network by combining *knowledge distillation* (Hinton et al., 2014) and *Bayesian optimization* (Snoek et al., 2012). The ability to accomplish such transfer learning is critical for training neural networks on large datasets and possibly in streaming context.

Acknowledgment

I would like to thank my supervisor, Prof Chua Tat Seng, for his insightful advice throughout the duration of this final year project. I reached a stumbling block halfway though my project when I cannot proceed with my original plan because of the unplanned lack of data. His direction at that time was very valuable and encouraged me to explore different alternatives for my project. He helps me to view my project goal from a bigger and more abstract perspective. This allows me to link together different new ideas I have for my project.

Special thanks to my friend Jie Fu who introduces me to various techniques and papers in machine learning and helps review my report. This project is inspired by his original idea of using Bayesian Optimization to optimize the selection of input order when approximating non-convex functions. My initial implementation did not produce the expected result, so I have to further explore areas that are related to non-convex functions. My final decision to experiment with training techniques of neural network will not be practically possible without him allowing to use his graphics processor (GPU). This enables me to conduct experiment by at least 15 times faster that if I were to use CPU alone.

I'm thankful for the developers of various Python libraries: Theano, Pylearn2, Spearmint, IPython and Numpy who make their libraries open source and well documented. Without these wonderful libraries, I would not have been able to perform my experiments due to the high complexity of the low level mathematics and functional parallel CPU and GPU programming required.

Contents

1 Introduction

Neural network is experiencing what Jürgen Schmidhuber referred as *second renaissance* as it gains popularity in recent years (Scardapane, 2015). Part of it is due to the advancement in computing power, especially the capability to use Graphic Processing Unit (GPU) to accelerate the computation. This allows the construction of complex and powerful neural networks. Another major reason is the availability and accessibility of the data to train the network. These new developments has led to the practical use and massive improvement in the the performance of neural networks. Deep neural networks are currently state of the art in complex tasks such as speech recognition and computer vision. It is no surprise that powerful learning techniques such as deep learning neural network is a popular research interest these days.

Despite these advancements, it still remains a challenge to train a neural network on a big dataset. When the size of the dataset exceeds the amount of memory, training the network becomes slow because a relatively slower disk storage has to be used. Furthermore, when the data comes intermittently, traditional way of training needs to wait for the complete data or repeat the training process every time a new batch is available. Otherwise, the generalization performance is severely affected due to catastrophic forgetting (Goodfellow et al., 2013). Meanwhile, the magnitude of data these days can easily reach terabytes in size. `1000 Genomes` project has 260 TB publicly available human genome data and `Internet Archive` shares 80 TB web crawl for research). Data streamed from social network or internet of things are also periodic by nature. Being able to train neural network effectively on these massive or intermittent data will tremendously improve the breadth of application of neural networks.

In this project, we explore techniques we can use to train neural network in an incremental manner, using only subset of the whole dataset at a time. Our contribution in the following report is the following:

- We propose a new approach, `Incremental KDMerge` in `section` 2.2 to train a neural network on different subsets of the dataset in a step-wise manner. `Incremental KDMerge` works by the application of knowledge distillation technique(Hinton et al., 2014) multiple times on different subsets of the data.

- We experiment with different initialization methods, i.e. how to transfer parameters from one neural network to another. In `section` 3 we demonstrate that step-wise parameter transfer works best in incremental training context.

- Knowledge distillation, as with most machine learning algorithms, has hyperparameters which can significantly affect the performance. We show in `section` 2.2 that *Bayesian optimization* (Snoek et al., 2012) is an effective technique to tune these hyperparameters.

In the following section, we give an overview of the main ideas behind neural networks and various model combination techniques. We describe the motivation

behind Bayesian optimization as a general optimization technique for any black box function. Then, we explore curriculum learning strategy (Bengio et al., 2009), which may be related to incremental learning.

1.1 Neural Network

The term neural network has its origins in attempts to find mathematical representations of information processing in biological systems (Bishop, 2007). Neural network resembles the way our brain processes information in that it acquires knowledge through a *learning process* and that the knowledge is stored via the interconnection strengths between neurons, known as synaptic *weights* (Haykin, 1994).

1.1.1 Multi-Layer Perceptron

Multi-layer perceptron (MLP) is one of the most common neural network architectures. It can be described as a series of functional transformations (Bishop, 2007) that tries to represent a mapping between input and the final output. Consider a linear combination of input variables $x_1, ..., x_D$ in the form

$$a_j = \sum_{i=1}^{D} w_{ji}^{(1)} x_i + b_j^{(1)} \tag{1}$$

where $j = 1, ..., M$ indicated the j-th *activation* units and the superscript indicates that these units are in the first *layer* of the network. The parameters $w_{ji}^{(1)}$ and $b^{(1)}$ are called *weights* and *biases* respectively. Each of them is then transformed using a differentiable *nonlinear activation function* $h(\cdot)$ to give

$$z_j = h(a_j)$$

The output from the previous layer can again be further transformed to give the following

$$a_k = \sum_{j=1}^{M} w_{kj}^{(2)} z_j + b_k^{(2)} \tag{2}$$

where $k = 1, ..., K$ is the total number of outputs. This transformation corresponds to the second layer of the network. The two layer MLP is shown in Figure 1 where the layer in the middle is called the hidden layer (with hidden units) and the last layer is called the output layer.

1.1.2 Convolutional Network

The network shown in Figure 1 is a common architecture used in practice where every activation unit is *fully connected* to the other units in adjacent layer. For specific tasks such as computer vision, however, a specialized kind of grid-like architecture called *convolutional* neural network is more suitable. (Bengio et al., 2014)

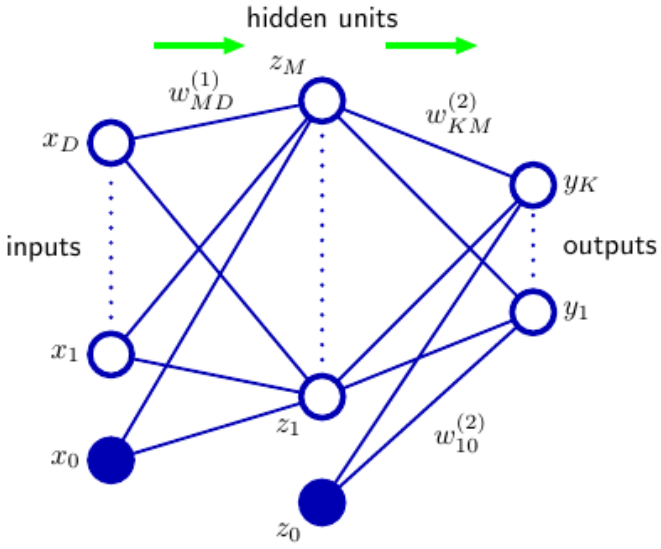

Figure 1: The input, hidden, and output variables are represented by nodes, and the weight parameters are represented by links between the nodes, in which the bias parameters are denoted by links coming from additional input and hidden variables x_0 and z_0 . Arrows denote the direction of information flow through the network during forward propagation (Bishop, 2007)

Convolution leverages three important ideas that can help a machine learning system: *sparse interactions, parameter sharing,* and *equivariant representations* (Bengio et al., 2014).

Sparse interactions allow convolutional network to exploit spatially-local correlation by enforcing local connectivity pattern between units of adjacent layers (Bengio et al., 2014). Every unit is now connected to only a subset of the units in the adjacent layer, as opposed to all units in a fully connected network. This lowers both the memory and number of computations required.

Parameter sharing involves constraining the weights of several hidden units to have the same value. Weight sharing not only reduced the number of free parameters that need to be learnt, but it also allows features to be detected regardless of their position in the visual field. This is known as the equivariant representation property of convolutional network.(Bengio et al., 2014)

Another important feature of convolutional network is *max-pooling* which is a form of linear down-sampling, where the input is partitioned into several non-overlapping partitions and a maximum value is outputted for each region (Bengio et al., 2014). Max-pooling provides further computational benefits and positional invariance, a property common in vision task.

Figure 2 shows a common architecture of convolutional network. Notice the sequences of alternating convolution and max-pooling layers to extract more and more abstract representation of the images. The final two layers are the traditional MLP that output a prediction.

1.1.3 Activation Functions

Multi-layer perceptron (MLP) powerful modelling capabilities come from the use of *non-linear* activation function. With non-linear activation function, MLP can model any non-linear functions by stacking multiple hidden layers, making it a *universal function approximator*. The activation function itself was developed to model the frequency of firing of biological neurons in brains .

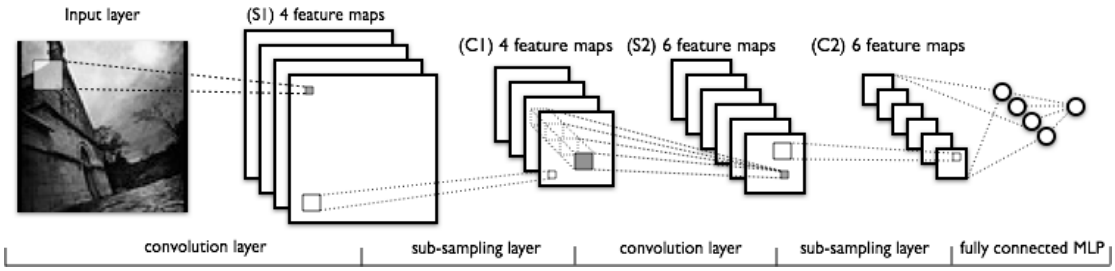

Figure 2: LeNet model of convolutional network, consisting of alternating convolution and max-pooling networks, with MLP(hidden layer + logistic regression) in the upper-layers.(LeCun et al., 1998a)

The choice of parameter values controls the degree and location of these non-linearities. With the appropriate choice of parameters, multi-layer neural networks can in principle approximate smooth functions, with more hidden units allowing one to achieve better approximations. (Bengio et al., 2014)

Some of the common non-linearities used are: *rectified linear unit, hyperbolic tangent, sigmoid, maxout, softmax* and *radial basis function*. In particular, the softmax activation function is normally used in the output layer to give discrete probability values for different classes of output. The probability that an input vector x is a member of a class i (a value of a stochastic variable Y) is represented as the following, (Bengio et al., 2014)

$$P(Y = i|x, W, b) = softmax(Wx + b) \tag{3}$$

$$= \frac{e^{W_i^x + b_i}}{\sum_j e^{W_j^x + b_j}} \tag{4}$$

1.1.4 Training

For a neural network to approximate a function effectively, its activation units need to have appropriate weight and bias values. The tuning of these values is accomplished by training the neural network on samples from the whole dataset. The most common training method is backpropagation algorithm where the error values for an input is propagated backwards to the previous layers. This allows the activation units of all layers to modify their weights in order to minimize the error backpropagated.

To define the error values, a loss or error function has to be defined. In the case of multi-class classification, a common loss function ℓ is the negative log-likelihood which is equivalent to maximizing the likelihood $\mathcal{L}$ of the dataset $\mathcal{D}$ under the model parameterized by θ (Bengio et al., 2014)

$$\mathcal{L}(\theta = \{W, b\}, \mathcal{D}) = \sum_{i=0}^{|\mathcal{D}|} log(P(Y = y^{(i)}|x^{(i)}, W, b)) \tag{5}$$

$$\ell(\theta = \{W, b\}, \mathcal{D}) = -\mathcal{L}(\theta = \{W, b\}, \mathcal{D}) \tag{6}$$

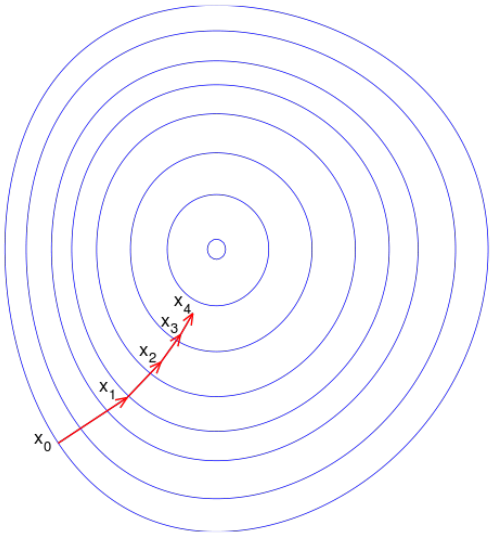

Figure 3: How gradient descent find the minimum point. Imagine the blue lines represent contour lines where the circle at the center is the lowest point we want to find. The red arrows show the direction and length in space that gradient descent chooses for four time steps from $x_0 \rightarrow x_1 \rightarrow ... \rightarrow x_5$.

A common and simple approach to minimize such non-linear function is the *gradient descent* optimization (Figure 3) where one takes steps proportional to the *negative* of the gradient. The weight values w are then updated as follows, (Bishop, 2007)

$$w^{\tau+1} = w^{\tau} - \eta \nabla E(w^{\tau})$$

where τ is the current step of the optimization. With enough iterations, w will be updated to values that minimize the corresponding loss function.

The non-linear activation function causes the objective function to be highly non-convex. This means that the parameters we obtained from the training algorithm may only *locally* minimize the function. To avoid getting stuck in a local minimum, we would intuitively train more than one neural networks, each with different initialization values. Then we can take the combined prediction of these networks. In the following, we give an overview of common methods to do so.

1.2 Combining Neural Networks

Combining predictions from different machine learning models are almost guaranteed to deliver more robust and accurate models without the need for the high-degree of fine tuning required for the single-model solutions (Abbott, 1999). Ensemble methods train different classifiers in a different way, and perhaps using different subset of examples. Techniques such as majority voting can then be used to combine the different predictions. From bias-variance perspective, combining models helps reduce the overall bias and variance of the combined models by lowering the variance of the flexible models (having low bias and high variance) and the bias of the inflexible models (having high bias and low variance) (Džeroski et al., 2009).

1.2.1 Ensemble Methods

The aim of ensemble methods is to combine the predictions of several base models to improve the generalizability over a single model. Commonly used ensemble methods are *bagging* and *boosting* because of its simplicity and improvement in practice..

Bagging

Bagging (short for bootstrap aggregation) is a voting method where base models are learned on different variants of the learning data set which are generated with bootstrap sampling, which is a sampling technique with replacement. Using these sampled sets, a collection of base models is learned and their predictions are combined by simple *majority voting*. Such an ensemble often gives better results than its individual base models because it combines the advantages of individual models. (Dietterich, 2000)

Boosting

Boosting is similar to bagging in that it uses voting to combine the prediction of various base models. However, rather than leaving the generation of next base models to chance as in bagging, boosting try to generate complementary base models by taking into account the mistakes of previous models. In other words, examples that are incorrectly predicted by the previous models are given more importance in the next base models so that the resulting base models will complement each other. (Džeroski et al., 2009)

1.2.2 Mixture of Experts

The combination of the base learners can also be governed by a supervisor learner that selects the most appropriate learner based on the input data, leading to the concept of mixture of experts (Jacobs, 1995). An additional gating network performs the division of the input space. Different neural network will then learn from these distinct divisions of input space. This method can even be extended by organizing the experts in hierarchical order. The outputs of different experts are then non-linearly combined by hierarchical supervisor gating networks.(Jordan and Jacobs, 1992).

1.2.3 Knowledge Distillation

Knowledge distillation was originally proposed by Hinton as a method to compress the knowledge of a bigger model into a smaller model. The motivation is that making predictions using ensemble of models is cumbersome and can be too computationally intensive for real time and mass deployment. Knowledge distillation allows the construction of smaller models that performs almost as good as the bigger models. This allows much faster deployment and runtime performance. (Hinton et al., 2014)

The main idea behind knowledge distillation is to use the class probabilities produced by the big and cumbersome model as the *soft targets* for training the small model. Since the soft targets are likely to have high entropy value, they tend to provide richer information per training case as compared to hard targets. With the same amount of training examples, soft targets allow the smaller model to learn much faster.

For relatively simple tasks where the model is mostly very confident about the correct output, the probabilities for other outputs may be far too small to be captured during training. To alleviate this problem, knowledge distillation raises the *temperature* of the softmax output. This reduces the conditional probability ratio between targets with high probability and targets with lower probability.

As described in equation 3, the softmax gives the output probability of each class for each example. Let a_T be the pre-softmax activation of the bigger model or teacher model. Then, knowledge distillation *relaxes* this output by temperature τ to give a softer output probabilities,

$$P_{T_i}^{\tau} = softmax\left(\frac{a_{T_i}}{\tau}\right) \tag{7}$$

Similar relaxation with same temperature τ is applied to the output of smaller model which we call student model to obtain P_S^{τ}. The student network is then set to minimize the following loss function,

$$\mathcal{L}_{KD}(W_S) = \mathcal{H}(y_t rue, P_S) + \lambda(\mathcal{H}(P_T^{\tau}, P_S^{\tau})) \tag{8}$$

where H refers to the cross-entropy and λ is a tunable parameter to balance both cross-entropies.

Knowledge distillation has been effectively used to build a new smaller model from existing bigger model while inheriting most of the generalization ability of the teacher. We demonstrate in `section` 3, that similar technique is also very effective to construct a more complex model from simpler ones.

1.3 Bayesian Optimization

Most machine learning algorithms involve the careful and time-consuming tuning of learning hyperparameters and model hyperparameters (Snoek et al., 2012). The choice of these hyperparameters can be the factor that determines the usefulness of a machine learning model. Historically, search techniques such as *grid search* or *random search* are employed to find these optimal hyperparameters. The drawback of these methods is the time consuming process these processes can take to find a reasonably good parameters. The stochastic nature of random search means we cannot predict how long we should wait until we have a reasonably good result. On the other hand, grid search involves manually handpicked values that are usually chosen based on experience and intuition. These values may miss certain unusual values, for instance values around parameters that give poor performance may normally be avoided for timesaving. A more principled way of choosing hyperparameters is to use Bayesian Optimization. Empirically, it has been found

to be a significantly superior method (Snoek et al., 2012) as compared to the former methods.

Bayesian optimization uses *Gaussian processes* as priors and construct a probabilistic model for $f(x)$. This probabilistic model can be used to decide where in X to next evaluate the function by integrating out uncertainty. The essential philosophy is to use all of the information available from previous evaluations of $f(x)$ and not simply rely on local gradient and Hessian approximations. This allows us to find the minimum of difficult non-convex functions with relatively few evaluations, at the cost of performing more computation to determine the next point to try. When the cost of evaluation is very expensive (as it is in most neural network training), this extra cost of computing the next evaluation is well justified. (Snoek et al., 2012)

1.3.1 Gaussian Process

The capacity of Gaussian process to represent a rich distribution on functions comes from *covariance function*, that encode all the assumptions we have about the functions we are modelling. Covariance function describes a measure of how two variables change together. There are classes of covariance function that are normally used such as the squared exponential and Matérn covariance functions (Rasmussen, 2006). In particular, the automatic relevance determination (ARD) Matérn 5/2 kernel is found to be suitable for practical optimization problems (Snoek et al., 2012).

1.3.2 Acquisition Functions

The role of the acquisition function is to guide the search for the optimum by balancing *exploration* (choosing point samples with high surrogate variance and hence potentially finding a new maximum points) and *exploitation* (choosing points with high surrogate mean to maximize the cumulative reward) trade-off. (Brochu et al., 2010)

Under Gaussian process prior, these acquisition functions depend on the model solely through its predictive mean function $\mu(x; \{x_n, y_n\}, \theta)$ and predictive variance function $\sigma^2(x; \{x_n, y_n\}, \theta)$. Some popular choices of acquisition functions are *probability of improvement*, *expected improvement*, and *upper confidence bound*. (Snoek et al., 2012)

After choosing the appropriate covariance and acquisition functions, one can then use Bayesian Optimization to efficiently sample the space containing the objective functions. Figure 4 shows an example of using Bayesian optimization to find a minimum value of a function in one dimension.

1.4 Curriculum Learning

Curriculum learning attempts to *ease* the learning process of machine learning algorithm by modifying the order and size of training examples. It involves creating sequences of training criteria, starting from ones that are easier to optimize, and

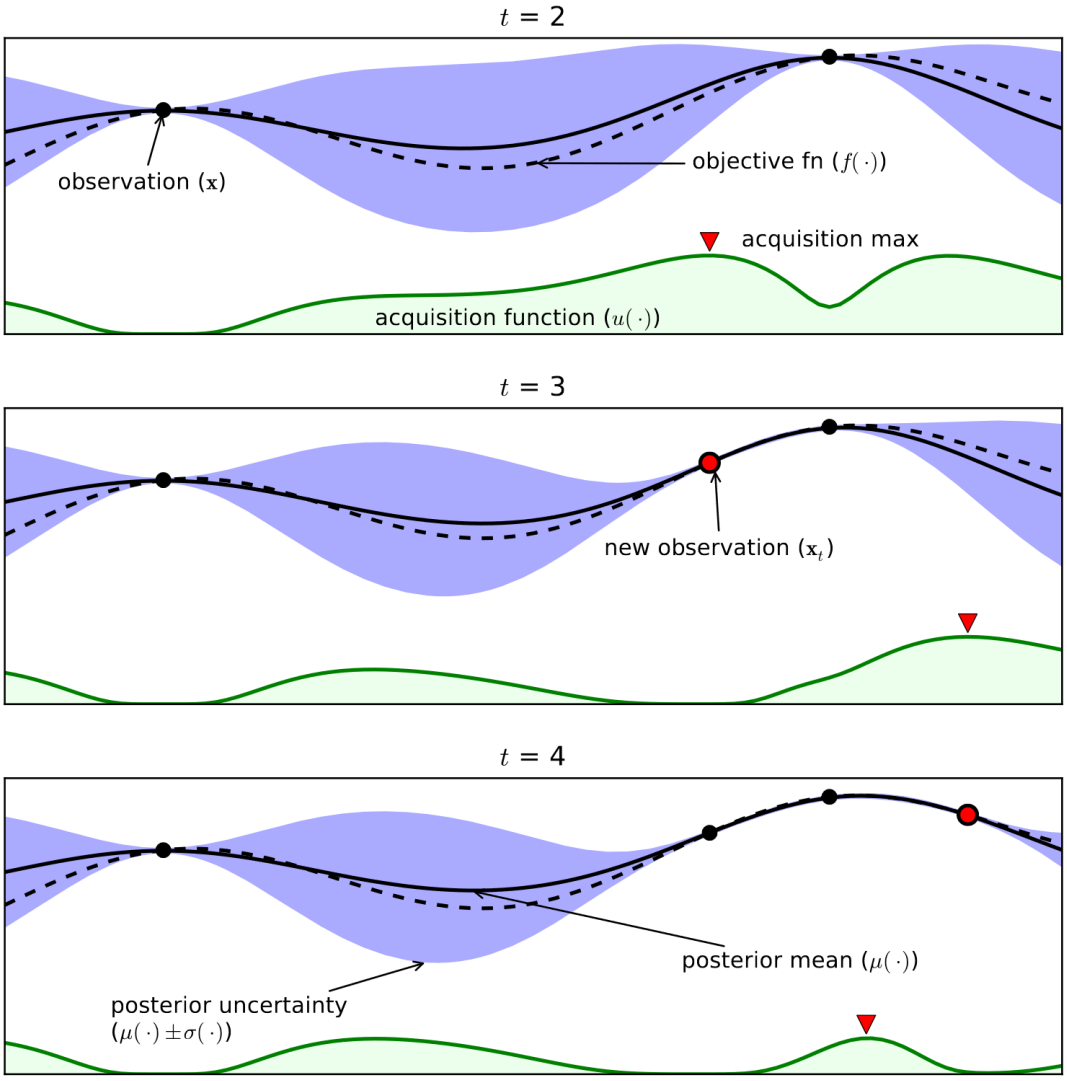

Figure 4: An example of using Bayesian optimization on a toy 1D design problem. The figures show a Gaussian process (GP) approximation of the objective function over four iterations of sampled values of the objective function. The figure also shows the acquisition function in the lower shaded plots. The acquisition is high where the GP predicts a high objective (exploitation) and where the prediction uncertainty is high (exploration)—areas with both attributes are sampled first. Note that the area on the far left remains unsampled, as while it has high uncertainty, it is (correctly) predicted to offer little improvement over the highest observation. (Brochu et al., 2010)

ending with the training criterion of interest (normally means the whole dataset). (Bengio et al., 2009)

1.4.1 Self paced learning

Curriculum learning advocates the ordering of input samples based on *easiness*. However, in practical situations, we are often not provided with a readily computable measure of the easiness of sample. An approach to tackle this issue is to use iterative *self-paced* learning algorithm where each iteration *simultaneously* selects easy examples and learns a new parameter vector, that is each iteration solve the following,

$$(w_{t+1}, v_{t+1}) = \underset{w \in \mathbb{R}^d, v \in \{0,1\}^n}{\operatorname{argmin}} \left(r(w) + \sum_{i=1}^{n} v_i f(x_i, y_i; w) - \frac{1}{K} \sum_{i=1}^{n} v_i \right) \tag{9}$$

where w is the parameter we are learning to minimize the loss function $f(\cdot)$ with regularizer $r(\cdot)$, v is a binary variable that indicates whether i^{th} sample is easy, and K is a weight that determines number of samples to include. As K is iteratively decreased, more samples are included until it involves all the examples. (Kumar et al., 2010)

(Kumar et al., 2010) has shown cases where $f(\cdot)$ is convex in w, which be treated as a biconvex optimization problem. For cases, where $f(\cdot)$ is non-convex (such as in neural network loss function) we require a different optimization method.

In `section` 4.4, we use another approach to define the latent variable v_i that determines how easy the i^{th} sample is and how it can be incorporated into neural network training algorithm.

1.4.2 Diversity

Placing too much emphasis on learning easier examples in self-paced learning may lead to overfitting to a data subset. Once some samples from a group have been learnt, self-paced learning will tend to prefer samples from the same group because they *appear* to be easy from what the current model has learnt. This may cause other easy examples from other group to be ignored and lead to worse generalization performance. In order to tackle this problem, diversity also needs to be factored in the process of self-paced learning. (Jiang et al., 2014)

Diversity suggests the choice of samples that are not similar and not clustered together. In (Jiang et al., 2014), another regularization term is introduced to the self-paced learning algorithm that places more weight on samples that are more diverse. This allows the model to achieve a better performance on two action datasets.

2 Methods and Implementation

We describe our approach to train neural networks in step-wise manner, using only a subset of the data in each training step. This eliminates the need to consume

or wait for the availability of the entire data.

2.1 Partitioning

Let D be the whole dataset and only D/N examples can be used at any time during training because of memory or computational constraint. D is first split into N equal partition: $D_1, D_2, ..., D_N$. Each of the partition is assumed to be independent and identically distributed. In other words, they come from the same probability distribution and are all mutually independent so we assume there is no preference in choosing one partition over another.

2.2 Incremental Merging

We train the neural network incrementally by training it multiple times using different partition of the data. At every training iteration, except the first one, the soft output from the recently merged network will be used as a guide to train a new model on the new data.

Algorithm 1 `Incremental KDMerge` repeatedly trains a new neural network M_i on new data partition D_i with the guide from most recent network M_{i-1}.

Input: Dataset D

Output: Final model M_{Final}

1: $D_1, D_2, ..., D_N \xleftarrow{partition} D$

2: $M_1 \xleftarrow{train} D_1$

3: **for** $i = 2$ to N **do**

4: $\quad M_i \xleftarrow[parameters]{transfer} M_{i-1}$

5: $\quad M_i \xleftarrow{KD} D_i + M_{i-1}$

6: **end for**

7: $M_{Final} \leftarrow M_N$

At initialization, we train neural network model M_1 on the first partition D_1 using backpropagation to minimize the negative log-likelihood loss function described in Equation 5,

$$\ell(\theta = -\sum_{i=0}^{|\mathcal{D}|} log(P(Y = y^{(i)}|x^{(i)}, W, b))$$

The best parameters for M_1 and other subsequent network M_i are selected based on the lowest validation error calculated at every epoch.

The next step is the transfer of parameters from the previous network to the new network. We copy the weight and bias values of the activation units from the

most recently trained network M_{i-1} to M_i. This approach is similar to performing supervised pretraining where each layer is trained greedily on the dataset, before performing the next stage of training (Larochelle et al., 2009). We find that for shallow fully connected network with only one hidden layer, copying all the weight and bias values from all layers (including the output log regression layer) of M_{i-1} to M_i works quite well. On the other hand, for deeper convolutional network, copying only the parameters from the first few convolutional layers works better than copying the parameters from all the layers.

In our implementation, M_{i-1} and M_i have the same structure, i.e. same number of hidden layers and hidden units, so *copying* the parameters from M_{i-1} to M_i works just fine. On the other hand, when M_{i-1} and M_i have different number of hidden layers and hidden units, transferring the parameters will not be as straightforward. One approach is to follow the *hint-based* technique of FitNets that adds an additional regressor on top of the *guided* layer M_i so that the number of outputs of the guided layer matches that of the *hint* layer of M_{i-1} (Romero et al., 2014).

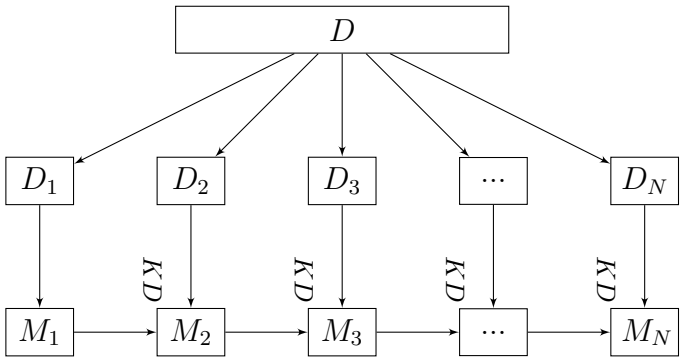

Figure 5: `Incremental KDMerge` first splits the dataset into N partition, depending on the constrained memory limitation. After training the first model M_1 on D_1, repeated iterations of parameter transfer and knowledge distillation train the new model M_i on the next data partition D_i, until all data partitions have been used. M_N represents our final neural network model.

Knowledge distillation is then used to train new network M_i on the next partition D_i, with the soft output from the recently merged network M_{i-1} acting as a hint or regularizer for M_i. To accomplish this, we set M_i to minimize the following loss function using stochastic gradient method,

$$\mathcal{L}_{KD}(W_{M_i}) = \mathcal{H}(y_t rue, P_{M_i}) + \lambda \cdot (\mathcal{H}(P^{\tau}_{M_{i-1}}, P^{\tau}_{M_i}))$$

where $\mathcal{H}(y_t rue, P_{M_i})$ represents the cross entropy between M_i and the true output for each example and $\mathcal{H}(P^{\tau}_{M_{i-1}}, P^{\tau}_{M_i})$ the cross entropy between the softened output of M_i and M_{i-1} for the same example.

This process of transferring parameters from previously merged network M_{i-1} to M_i and then training M_i on D_i with knowledge distilled from M_{i-1} is repeated $N-1$ times. Network M_N will be final model which is able to generalize all the examples from D reasonably. Figure 5 shows the working of algorithm 1 graphically.

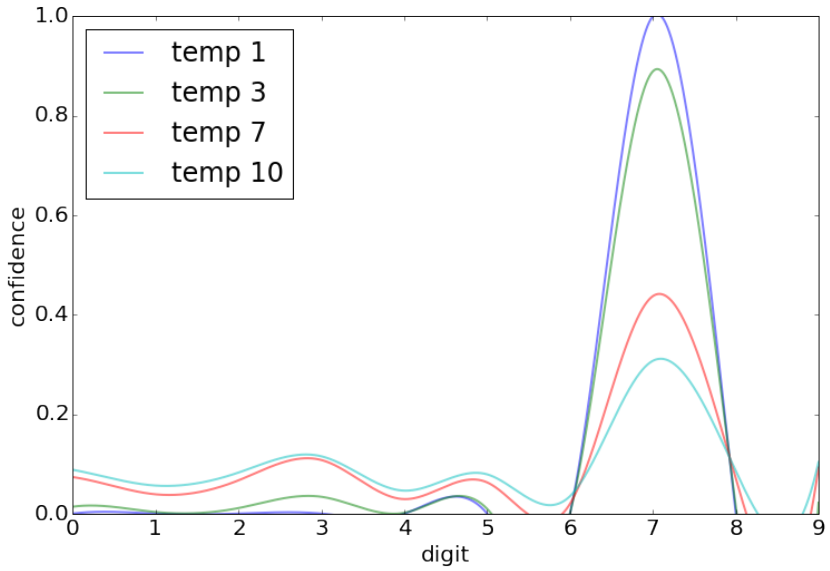

Figure 6: How the softmax output for an example varies with temperature. In this case, the model has a high confidence that the example is digit 7.

Tuning Knowledge Distillation

The effectiveness of knowledge distillation depends on setting appropriate values of temperature and relative teacher weight. These values, however, are empirical at nature (Hinton et al., 2014) because it is data and model dependent.

Temperature τ determines how much we want to relax the softmax output from the teacher model. Using high temperature allows the student network to better capture the similarity or differences between an output label with another output label. However, too high a temperature may result in differences between distinct output labels to be too subtle for student to learn effectively. Figure 6 shows the effect of different temperatures on the output from softmax. We can see how increasing the temperature distributes the confidence value from labels with high confidence to labels with lower confidence.

The weight of teacher λ reflects how much the student intends to learn from the teacher soft output. While learning the hard output from the actual examples guarantee that the correctness of the features learnt, the student may not have many enough examples to learn from because each partition only consists of D_i/N examples. On the other hand, the teacher has learnt from the whole examples in the partition and hence are more knowledgeable about the features that identify the whole partition. By using higher values of teacher weight, the student can learn knowledge about data from previous partitions.

To determine suitable values of temperature and relative teacher weight for our particular dataset and model, we use Bayesian optimization based on Gaussian processes. This allows us to shorten hyperparameter search time as Bayesian Optimization is designed to minimize the number of evaluations in finding the minimal value of any black-box function.

We use *Spearmint* library with Matérn 5/2 covariance kernel and maximum expected improvement (Max EI) acquisition function in our test. Max EI is found to be suitable in many optimization tasks and therefore it is our choice of acquisition function. (Snoek et al., 2012)

3 Experiment Results

We empirically test our `Incremental KDMerge` approach to training neural network using two publicly available datasets: MNIST (LeCun et al., 1998b) and CIFAR-10 (Krizhevsky and Hinton, 2009).

In order to simulate the problem of memory and computational constraint to use all the dataset at once, we divide the training dataset into equally sized and mutually exclusive partitions of 2 and 4. When dividing the dataset into k partitions, we assume that we can only use N/k of all available training examples during training. Therefore, in order for our model to observe all training examples, training has to be done in multiple iterations on different subsets of the data. Naturally, larger value of k is needed when the size of the training set is larger.

There are many possible approaches to train different partitions of the training examples. We use the following as a baseline comparison to our proposed `Incremental KDMerge`:

- ***split***: For each k partition, randomly select $1/k$ examples, to get a total of $k * (1/k) * (N/k) = N/k$ examples, which still fits our constraint of only having N/k at any time during training.

- ***average***: Similar to split in that we randomly select $1/k$ examples for every partition as training examples. However, *average* initialized the network differently. We first train k different network on the k different partitions of the training dataset. The average of the parameter values of these k models are then transferred to the new network as an initialization step.

- ***step***: Sequentially train the model on the different partitions of the dataset. The parameter values of the network M_{i-1} is transferred to the new network M_i being trained. This acts as a form of supervised pretraining.

step kd is our proposed method to train the model sequentially on different dataset partitions, similar to the *step* method. However, after transferring the parameter values from the most recent model M_{i-1}, we train the new network using knowledge distillation approach that uses soft output from M_{i-1} as a guide during training.

The ideal training outcome will be based on ***no split*** training, where the neural network is trained directly on the whole dataset $\mathcal{D}$, which we assume to be too huge or periodic and therefore has to be consumed partially.

3.1 MNIST

MNIST is a digit recognition dataset, as shown in Figure 7, consisting of handwritten digits of 60,000 examples of training set and 10,000 examples of test set. The digits have been size-normalized and centered in a fixed-size image of `28x28`. In our experiment, 10,000 of the training set is used as a validation set.

We use multi-layer perceptron to model our neural network with one hidden layer of `250` units, learning rate of `0.300`, L2 regularization of `0.004`. We tried

using more and fewer hidden layers and hidden units but the performance difference is not that substantial so we decided to stick with a relatively small network that still gives considerably good performance, while allowing use to experiment with different training methods. The other hyperparameters such as learning rate and L2 values are also found by empirical grid search and we again chose values that give reasonable result. Same network structure and hyperparameters are then used for different experiments on MNIST to make sure that any differences observed are not due to different learning rate, L1 and L2 regularizer or network architecture.

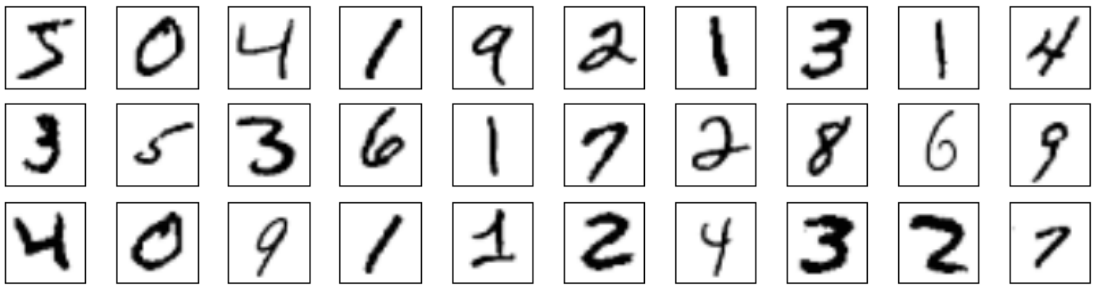

Figure 7: Sample images from MNIST dataset.

Figure 8 shows the validation error vs number of training epoch for datasets partitioned into 2. We include the *no split* result as a comparison if we are to train on the whole dataset at once.

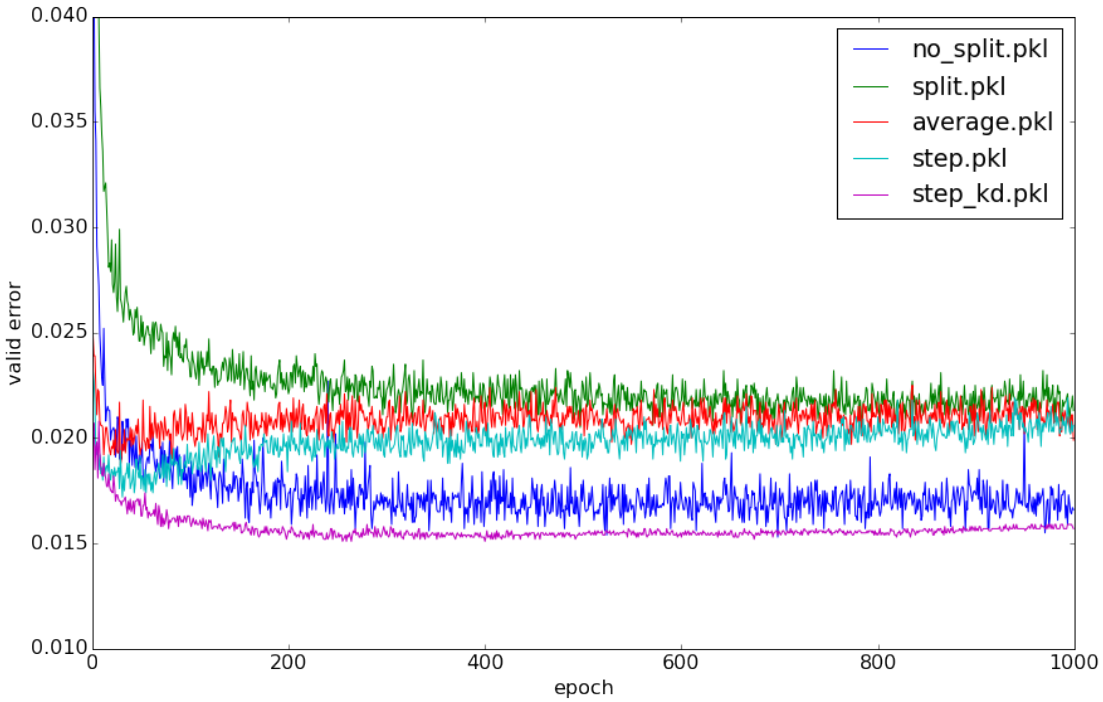

Figure 8: MNIST validation error on 2 partitions of the training examples.

step method performs better than *average* and *split* methods and is worse than

the *no split* method as expected. With knowledge distillation, we can significantly improve *step* as seen from *step kd* curve that is lower than *step*. Surprisingly, *step kd* method consistently has better validation error than *no split* method.

In order to ensure, that our proposed methods are not biased towards validation error — because we pick the teacher based on the validation set, we will end up using the validation set multiple times which may introduce some bias towards the validation set — we test our best performing model on the test set as well. Table 1 shows the test error result.

Training method	Validation error	Test error
no_split	0.0153	0.0159
split	0.0205	0.0223
average	0.0187	0.0203
step	0.0172	0.0192
step_kd	**0.0151**	**0.0155**

Table 1: MNIST validation and test error on 2 partitions of the training examples.

The test error result is relatively consistent with the validation error result and *step kd* appears to have the best generalization performance for classifying digits in MNIST even though the network only see half of the training examples during each training steps.

Figure 8 shows how *step kd* performs when the MNIST training dataset is further split into 4 partitions. The steep fall in validation error for *step* and *step kd* corresponds to the switching of examples from one partition to another (after it has converged on the previous partition). Similar to the result for 2 partitions, *step* has better lowest validation error as compared to *average* and *split*. Although the validation error at the last training stage rises beyond that of average and split (in other words, we need to perform early stopping for *step* method).

Our proposed *step kd* method appears to perform the best amongst the other methods that only use limited examples at any time during training. Unlike the result for 2 partitions, this time the minimum validation error of *no split* method is, as expected, better than *step kd* by about 12 %.

We also notice the increase in validation error with all the methods as we further partitions the training examples. This increase is lower in *step kd* than in other methods. Further partitioning from 2 into 4, results in increase in minimum validation error of 15% for *step kd*, 31% for *step*, 49 % for *average* and 41 % for *split*.

Table 1 shows the test error result to ensure that the error rate is not biased towards validation set.

We note that baseline results from deep learning network with multiple hidden layers has error rate of 1.5% or below (Benenston, 2015), and our result is therefore still comparable. Further improvement can be obtained by using deeper network or convolutional network as compared to multi-layer perceptron, and using validation set as part of the training set.

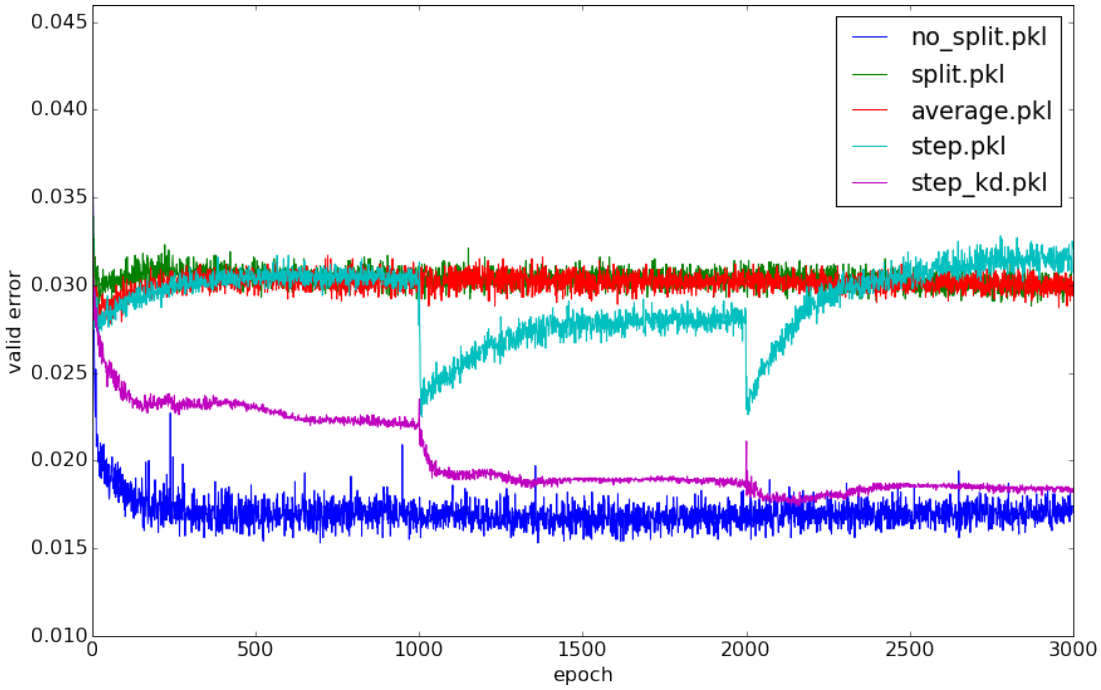

Figure 9: MNIST validation error on 4 partitions of the training examples.

Training method	Validation error	Test error
no_split	0.0153	0.0159
split	0.0285	0.0298
average	0.0278	0.0293
step	0.0225	0.0231
step_kd	**0.0174**	**0.0171**

Table 2: MNIST validation and test error on 4 partitions of the training examples.

3.2 CIFAR-10

The CIFAR-10 are labeled subsets of the 80 million tiny images dataset. Some examples are shown in Figure 10. It consists of 60000 32x32 colour images in 10 classes, with 6000 images per class. There are 50000 training images and 10000 test images. The test batch contains exactly 1000 randomly-selected images from each class. The training batches contain the remaining images in random order, but some training batches may contain more images from one class than another. Between them, the training batches contain exactly 5000 images from each class. For our experiment, we use the last 5000 images in the training set as a validation set. We also perform global contrast normalization and whitening on the images so that the dataset will have an identity covariance.

We use convolutional neural network with 2 layers convolutional layers and one hidden fully connected layer of 500 units as the architecture. Unlike MNIST

dataset, where the images are grayscale and the object of interest has similar shape and is clearly separated from the background, CIFAR-10 is made up of natural images with 3 color channels and more varied shapes. Using fully connected non-convolutional layers give us poor classification performance so we use the convolutional network instead. Convolutional network performs much better than fully connected network when the dataset has spatial features that can be exploited, as in the case of natural image classification.

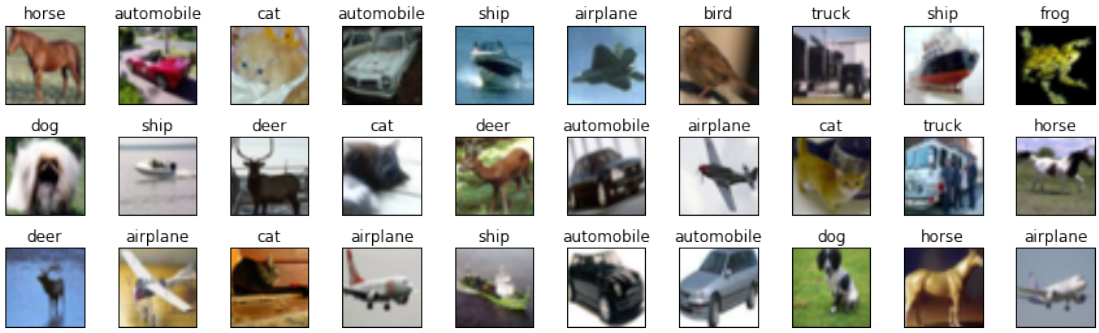

Figure 10: Sample images from CIFAR-10 dataset.

The validation error result for different training methods on CIFAR-10 split into 2 partitions is shown in Figure 11. The *no split* result is the case where no partition is done, and the network is trained on the whole dataset. This serves as an ideal scenario.

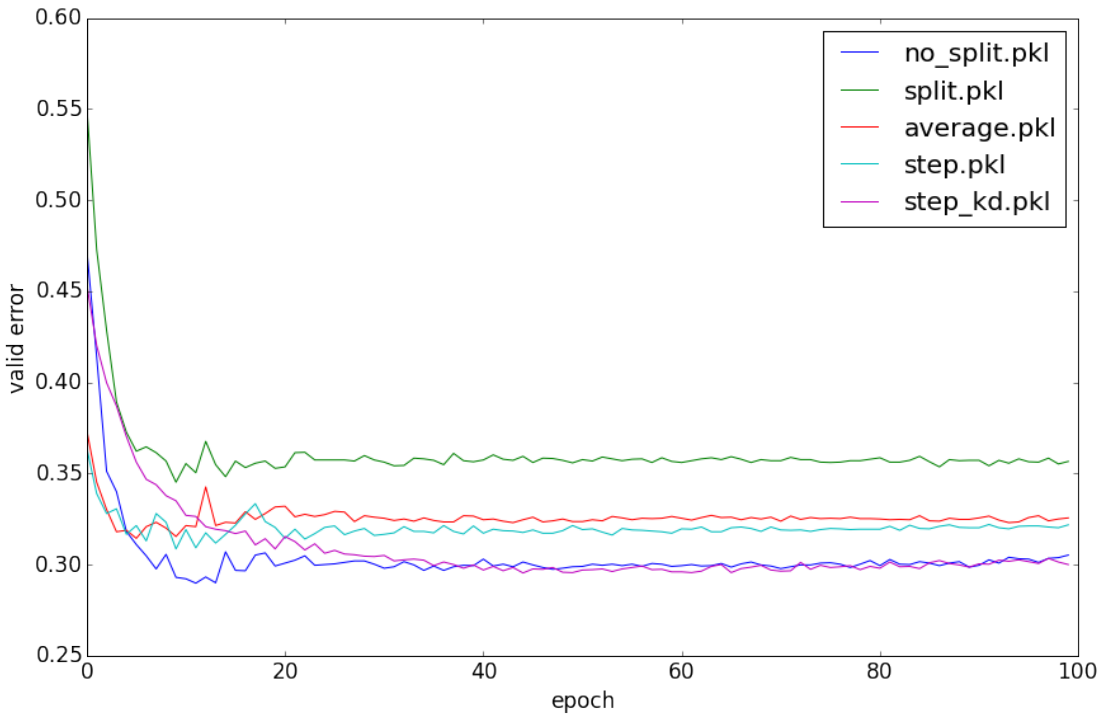

Figure 11: CIFAR-10 validation error on 2 partitions of the training examples.

The validation error for *step kd* appears to be the best, followed by *step*,

average and *split*. When looking at the best validation error obtained, *no split* is considerably better than *step*. However, when the neural network converges and stabilizes, *step kd* and *no split* appears to have similar performance.

Table 4 shows the test error result for the different methods on CIFAR-10 split into 2 partitions.

Training method	Validation error	Test error
no_split	0.2898	0.3007
split	0.3452	0.3586
average	0.3144	0.3290
step	0.3086	0.3274
step_kd	**0.2954**	**0.3102**

Table 3: CIFAR-10 validation and test error on 2 partitions of the training examples.

We further partition CIFAR-10 training examples into 4 to demonstrate how *step kd* scales up. Figure 12 shows the validation error result as training epochs progresses until convergence.

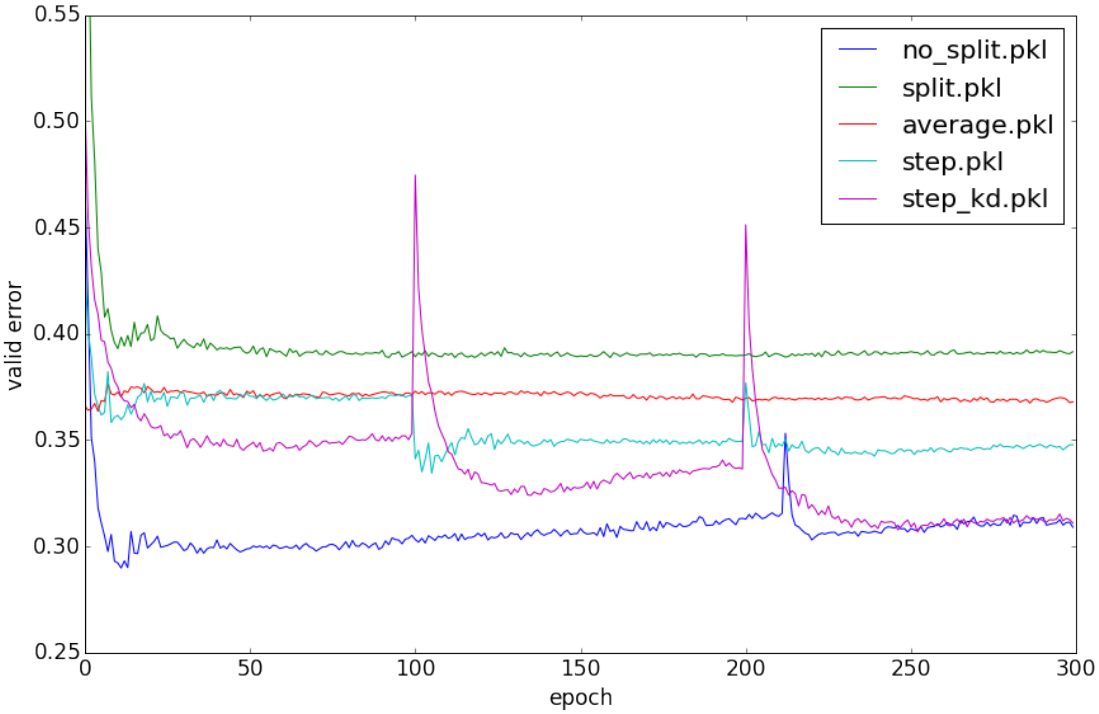

Figure 12: CIFAR-10 validation error on 4 partitions of the training examples.

Results similar to the expriments on MNIST can be observed from the figure, with *step kd* having the lowest validation error followed by *step*, average and split. At the end of the converge (nearing 300 epochs), performance of *step* kd is similar to that of *no split*.

The increase in validation error as we increase the number of partitions from 2 to 4 on CIFAR-10 is the least on *step kd* with 3.9 %. For *step*, *average* and *split*, the validation error is increased by 8.4 %, 15.6 % and 12.6 % respectively.

Table 4 shows the corresponding test error on 4 partitions of CIFAR-10 training examples. Our base performance of 28% error rate is noticeably worse than other benchmark results of 22% or less (Benenston, 2015). To achieve such performance, we may need a deeper and wider convolutional convolutional network. However, we do not attempt that because of our limited time budget to train and tweak the network. However, we believe that the trend in the performance of different training methods we present here will be applicable to more complicated and bigger network as well. We fix the hyperparameters such as learning rate, regularization parameters, and layer structures which serve as a control as we vary the training methods.

Training method	Validation error	Test error
no_split	0.2898	0.3007
split	0.3888	0.4056
average	0.3636	0.3682
step	0.3344	0.3495
step_kd	**0.3068**	**0.3225**

Table 4: CIFAR-10 validation and test error on 4 partitions of the training examples.

4 Discussion

We propose incremental $KD\ Merge$ in the scenario where we are constrained in the number of examples we can use during training. This constraint means we have to choose subset of the whole training examples during training. The training itself can be run in multiple stages on mutually exclusive subsets of the dataset, or a single stage on a subset of the dataset that still fits the constraint.

4.1 Supervised Pretraining

Our experimental results on MNIST and CIFAR-10 on both 2 and 4 partitions show that *step* method works better in general as compared to *split* and *average*. This implies that initialization from a single model is better than random initialization or initialization from the average of smaller models.

The initialization of a neural network can be linked to the task of pretraining the network. Larochelle et al. (Larochelle et al., 2009) explored various pretraining techniques such as stacked restricted Boltzmann machines (SRBM) network, stacked autoassociators (SAA) network, stacked logistic autoregressions network and greedy supervised pretraining network. His conclusion is that network with

pretraining performs better in general than that without pretraining. Although unsupervised pretraining is better than supervised pretraining.

Our methods of transferring parameters from existing to a new network can be seen as a form of supervised pretraining. This explains why *average* and *step* has better performance than *split* which uses random initialization (the suggested ranges depend on the activation function used (Glorot and Bengio, 2010)).

On the other hand, we are not certain why initializing the network multiple times with the parameters from previous model is better than initializing it once with the average of smaller models. We hypothesize it is due to *knowledge dilution* that happens when parameters from too many models are combined together. The optimal weight values from one model is overridden by the poor weight values from the other models.

4.2 Catastrophic Interference

It is clear from the validation error plots that *step* method quickly reaches a good performance but then it slowly deteriorates and loses it generalization ability. This is a known problem in neural network called catastrophic forgetting (Coop and Arel, 2012; Goodfellow et al., 2013) where the model slowly forgets what it has learnt once it is exposed to new training examples.

Figure 9 clearly shows this behaviour. For first few epochs after being shown examples from new partition of the dataset, network trained with *step* method very rapidly reduces the validation error. We can attribute this to the effective initialization of the parameters based on the previously trained model. However, the validation error gradually starts to increase when it has seen the new examples more and more times, meanwhile forgetting the pattern learned from examples from the previous partition.

The proposed *step kd* method can be seen as a regularizer that lessens this catastrophic interference. We can see from Figure 9 that with *step kd*, the initial drop in validation error is not as fast as that of *step*. However, the validation error does not rise once it falls because the new network also has to learn from the relaxed output from the previous model. This balances the knowledge that it has to learn from both the new examples and old examples.

Additionally, we can think of the previous network as a compressed representation of the data it was trained on. `Incremental KDMerge` works even if we only use one data partition because the previous model roughly represents the other data partitions. With this approach we do not need to have the whole data to be available during training.

4.3 Effectiveness of Bayesian Optimization

Using Bayesian optimization, we hope to find optimal setting of `temperature` and `lambda teacher` in as few tries as possible. Table 5 shows the number of evaluations performed to find the hyperparameters that gives minimum validation error, the mean and noise of the functions we are optimizing, and how sensitive the validation error is to changes in `temperature` and `lambda teacher`.

Bayesian optimization allows us to find good hyperparameter settings in 70 evaluations or less for the 3 experiments. For `MNIST 2`, at the 40^{th} evaluation, a validation error of 0.0148 has actually been achieved, which is marginally close to the best of 0.0146. Arguably, a better minimum might still be achieved if we run the optimization for longer period. However, we can see that by the 70^{th} evaluation, we can obtain a validation error that is considerably better than the estimated mean. For `CIFAR-10 2` the minimum is 11% lower than the estimated mean, although for `CIFAR-10 4` it is only 1.4% better than the estimated mean.

Experiment	Eval (Min, Total)	Min (Max) Error	Est. Mean (Noise)	Sensitivity^{-1} (τ, λ)
MNIST 2	146, 238	0.0146 (0.0214)	0.0154 ($1.80E^{-09}$)	1.90, 1.87
MNIST 4	41, 115	0.0191 (0.0223)	0.0200 ($5.66E^{-10}$)	1.98, 1.98
CIFAR-10 2	69, 127	0.2942 (0.3904)	0.3315 ($2.86E^{-06}$)	1.80, 1.17
CIFAR-10 4	45, 52	0.3080 (0.3276)	0.3123 ($6.70E^{-08}$)	0.84, 1.65

Table 5: No. of evaluations performed to find the optimal hyperparameters. For example, the min validation error of 0.0146 for `MNIST 2` is obtained after performing 145 evaluations out of the total 238 evaluations. *Spearmint* also additionally shows the average and noise of the function being optimized, as well as how sensitive the function to each of the hyperparameters.

It is also interesting that for MNIST, `temperature` and `lambda teacher` are about equally important while for CIFAR-10 `lambda teacher` is more important for 2 partitions but `temperature` is more important for 4 partitions.

We make an assumption that for the 3 steps of merging with 4 partitions, all merging will have the same optimal values of τ and λ. This might not actually be the case. Using different τ and λ for each merging may give even lower validation errors for `MNIST 4` and `CIFAR-10 4`. However, to perform Bayesian optimization on each of the merging will require substantially longer time so we do not carry it out.

Bayesian optimization also works better when we can better specify a good range of the hyperparameters. This will significantly speed up the the optimization processes as there is less parameter space to explore. In our experiment, we are not really certain of the correct range to set, so we set a quite wide range, for instance $0.1 \leq \lambda \leq 0.9$.

Furthermore, for our Bayesian optimization experiments, we also set our neural network to terminate earlier than what we show in `Section` 3, again to speed up the optimization process. Therefore, the hyperparameters we obtained from Bayesian optimization may only be *close* to the optimal values.

4.4 Training with Selective Inputs

`Incremental KDMerge` assumes that partitions of the data are equally useful to the learning process. In curriculum learning, however, (Bengio et al., 2009) suggests that some data allows a learning model to learn faster than the other. His curriculum learning method starts training a neural network with examples that

are *easier* to learn then gradually include more and more *hard* examples. He empirically shows on simple shape dataset and language modelling that this technique gives a model that generalizes better. Nevertheless, the determination of the *easiness* of examples still need to be carried out manually.

Inspired by the discovery that some examples are more useful than the other at different training stages, we try partitioning the dataset into easy and hard partitions. However, rather than classifying easy and hard examples with rules specific to the dataset, we use the neural network's confidence output to measure the easiness of different examples as shown in Algorithm 2. We end up with two partitions of easy and hard examples. Figure 13 and 14 shows 30 hardest and easiest examples respectively, as measured by the confidence of the model. In the case of MNIST, we can see that confidence seems to give a reasonably good measure of the sample easiness.

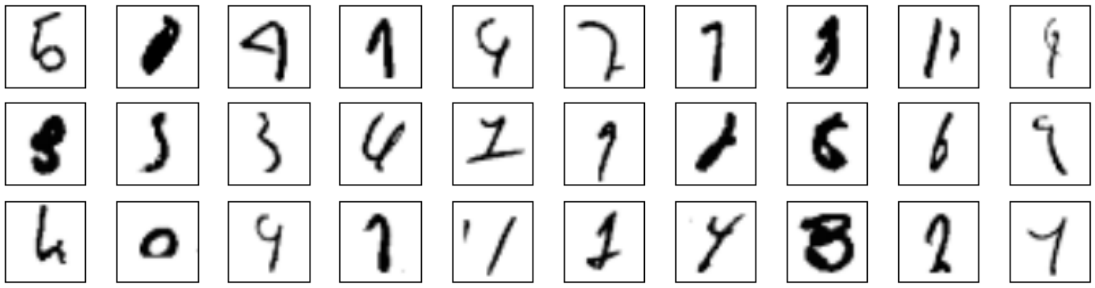

Figure 13: 30 hardest examples from MNIST, according to model confidence.

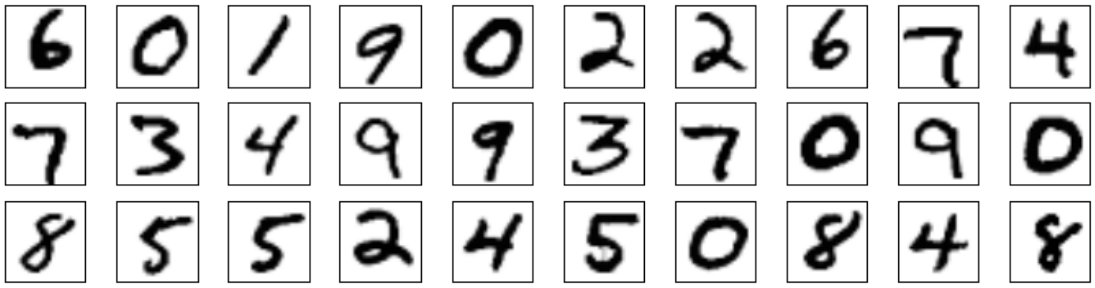

Figure 14: 30 easiest examples from MNIST, according to model confidence.

We trained a neural network on MNIST and CIFAR-10 datasets on only either $Partition_{easy}$ or $Partition_{hard}$, using the same hyperparameters in Section 3. Figure 15 and 16 shows the result on MNIST and CIFAR-10 respectively.

We can observe that using only the easy or hard examples gives a reasonable performance. For MNIST, training the network on the hard examples surprisingly gives similar validation error to training on all examples. Using the easy examples, however, result in significant decline in performance. The converse can be seen from result on CIFAR-10. Training on the hard examples actually result in substantially worse performance than training on easy examples.

The reason why hard examples work best on MNIST while easy examples work best on CIFAR-10 is not very clear. We hypothesize that this is related to the

Algorithm 2 `Confidence-based partitioning` creates 2 partitions from dataset D: ($Partition_{hard}$, $Partition_{easy}$), while ensuring that diversity is maintained in each minibatch.

Input: Dataset D
Output: Two dataset partitions: $Partition_{hard}, Partition_{easy}$

```
Train a baseline model M_baseline ← D
Confidence(x) ← max(softmax(x))
X_class_i ← examples belonging to class i in ascending confidence order
for all minibatches do
    for all examples in the minibatch do
        Pop an example from X_class_i and replace example_class−i
    end for
end for
Partition_hard ← 1st half of the dataset
Partition_easy ← 2nd half of the dataset
```

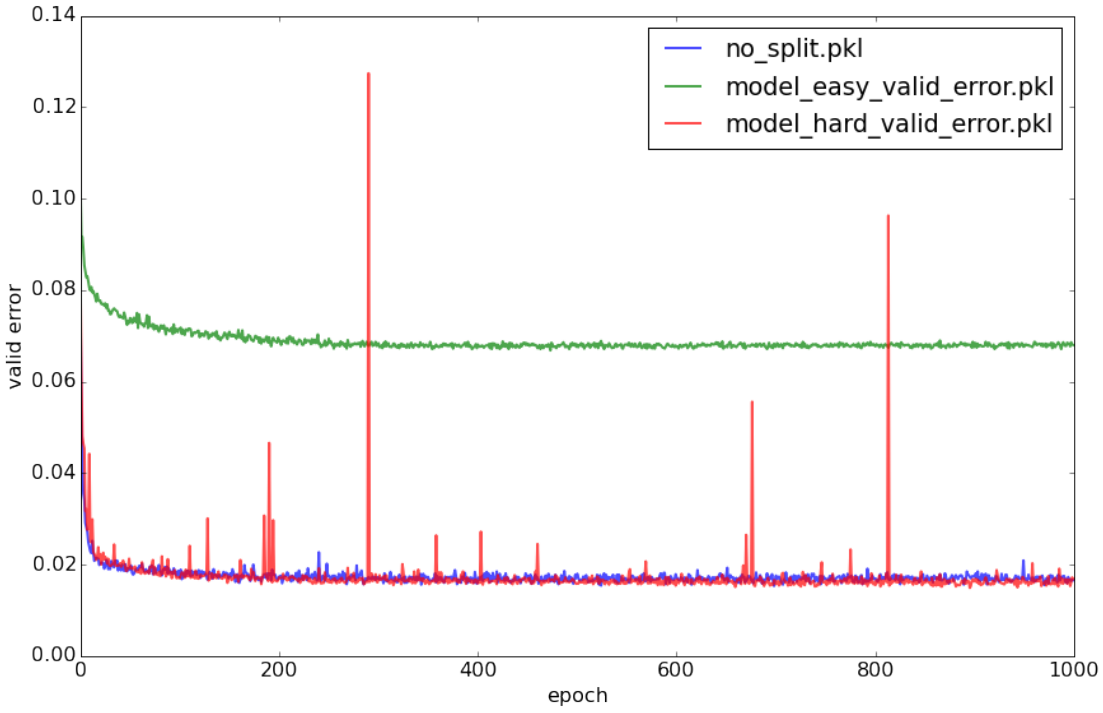

Figure 15: Validation error for model trained on MNIST using all examples (*no split*), hard partition only and easy partition only

model complexity. Classifying digits on MNIST is much simpler than classifying natural images on CIFAR-10. The neural network architecture we use in this experiment may be good enough for learning hard examples from MNIST but it may not be robust enough to learn from hard examples from CIFAR-10. We can see from here that not all examples are equally useful for learning, similar to the hypothesis proposed in curriculum learning.

This brings another possible approach to training neural networks using only

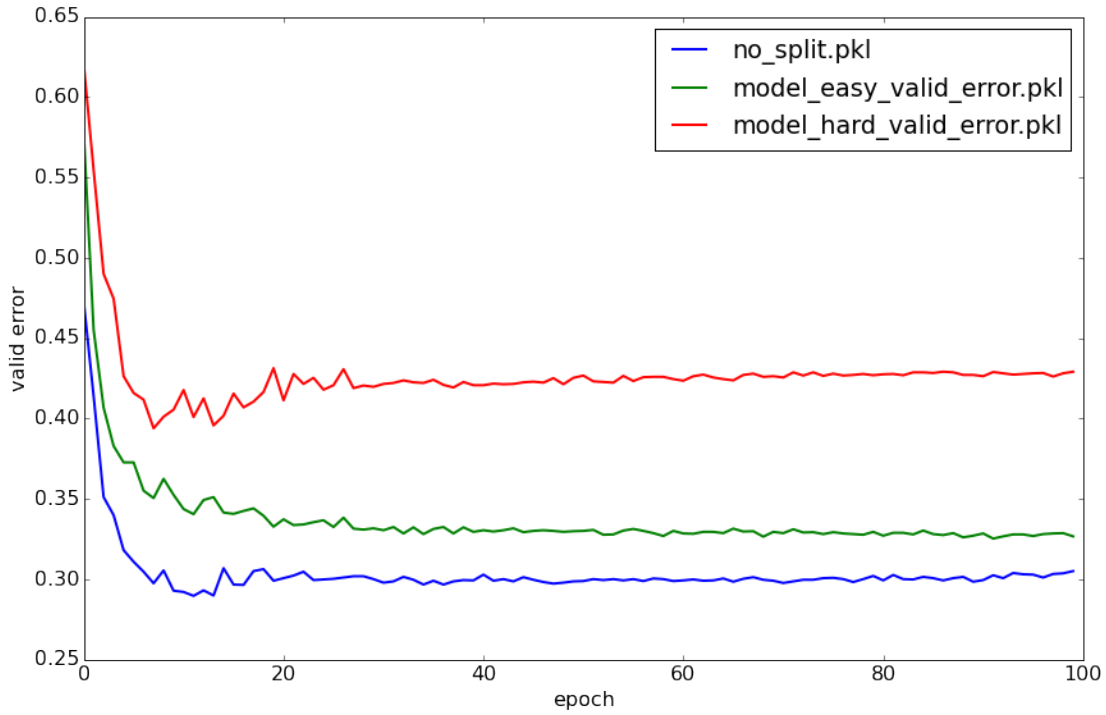

Figure 16: Validation error for model trained on CIFAR-10 using all examples (*no split*), hard partition only and easy partition only

subset of the data at a time. As opposed to `Incremental KDMerge` that tries to scan all examples by training the network in several steps, perhaps a *careful selection* of examples from the data can be used instead.

5 Related Work

Our proposed `Incremental KDMerge` is related to FitNets (Romero et al., 2014) in that the training of neural network is divided into two stages. The first stage involves initializing the parameters of the new network by using existing parameters from the teacher network and the second stage involves learning from both the examples and relaxed output from the teacher network. In FitNets case, however, both the teacher and the new network learns from the same dataset, and initialization with the help of the teacher helps the new network to have deeper network and better generalization performance. In this report, we have shown that the teacher and student network can be trained on a different dataset. Also, by repeatedly applying this two-stage training scheme, to cover the whole subsets of the data, we can gradually improve the performance of the network, as newer and more interesting labelled examples are seen. In a way, our method can be seen as repeated application of FitNets on different subsets of the dataset.

The knowledge distillation(Hinton et al., 2014) technique we use is also related to model compression (Bucilua et al., 2006) in that it aims to compress the performance of a big and cumbersome model (which may involve an ensemble of hundred

or thousand learning models) into smaller, faster models without significant loss in performance. Our method shows that knowledge distillation can actually works for extracting knowledge from models of similar complexity (complex in terms of the amount of data it has learned from).

We can also view `Incremental KDMerge` as a form of transductive transfer learning (Pan and Yang, 2010) where the source and target domains are different but related, and the source and target tasks are the same. The source and target domains are different because we are trying to mimic the performance of a network trained on the whole dataset (the target domain) by training the network multiple times on different subsets of the data (the source domain consists of subsets of target domain).

6 Conclusion

When training a neural network on a huge or intermittent dataset, the training method may need to adapt to only using subsets of the data at a time. We have shown that `Incremental KDMerge` is an effective method for this task. The network is trained stepwise, using mutually exclusive subsets of the data for each step. `Incremental KDMerge` makes use of knowledge distillation to balance the knowledge learnt from previous model and the new data. Using this method, we can achieve generalization performance that is comparable to that obtained by training the neural network on a whole dataset. This allows one to to train a neural network on a bigger and periodic dataset.

7 Further Work

We stated the challenge of training a neural network on a huge or intermittent dataset. However, due to time constraint we did not experiment with the actual huge or periodic dataset. However, using smaller dataset allows us to train a baseline model with which we compare the performannce against our proposed method. Future work can perhaps focus on using `Incremental KDMerge` on a real huge dataset such that training using subsets of data is really unavoidable.

Different ways of partitioning and selecting examples also require further research. We partition the data into mutually exclusive subset. Perhaps better performance can be obtained by using overlapping data partitions. Our preliminary investigation with ordering data based on confidence shows that 'easy' and 'hard' examples do matter when training a neural network. An exploration on partitioning the data based on the mix of easy and hard partitions can give insight on the role 'easiness' plays in the training of neural network.

References

Multilayer perceptron, January 2015. URL `http://en.wikipedia.org/w/index.php?title=Multilayer_perceptron&oldid=642315718`. Page Version ID:

642315718.

Dean W. Abbott. Combining models to improve classifier accuracy and robustness. In *Proceedings of Second International Conference on Information Fusion, Fusion'99*, volume 1, pages 289–295, 1999. URL http://isif.org/fusion/proceedings/fusion99CD/C-169.pdf.

Rodrigo Benenston. Classification datasets results, 2015. URL http://rodrigob.github.io/are_we_there_yet/build/classification_datasets_results.html.

Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. Curriculum learning. In *Proceedings of the 26th annual international conference on machine learning*, pages 41–48. ACM, 2009. URL http://dl.acm.org/citation.cfm?id=1553380.

Yoshua Bengio, Ian J. Goodfellow, and Aaron Courville. *Deep Learning*. MIT Press (preparation version 22/10/2014), 2014. URL http://www.iro.umontreal.ca/~bengioy/dlbook/front_matter.pdf.

Christopher Bishop. *Pattern Recognition and Machine Learning*. Springer, New York, October 2007. ISBN 9780387310732.

Eric Brochu, Vlad M. Cora, and Nando De Freitas. A tutorial on Bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning. *arXiv preprint arXiv:1012.2599*, 2010. URL http://arxiv.org/abs/1012.2599.

Cristian Bucilua, Rich Caruana, and Alexandru Niculescu-Mizil. Model compression. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 535–541. ACM, 2006. URL http://dl.acm.org/citation.cfm?id=1150464.

Robert Coop and Itamar Arel. Mitigation of catastrophic interference in neural networks using a fixed expansion layer. In *Circuits and Systems (MWSCAS), 2012 IEEE 55th International Midwest Symposium on*, pages 726–729. IEEE, 2012. URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6292123.

Thomas G. Dietterich. Ensemble Methods in Machine Learning. In *Multiple Classifier Systems*, number 1857 in Lecture Notes in Computer Science, pages 1–15. Springer Berlin Heidelberg, 2000. ISBN 978-3-540-67704-8, 978-3-540-45014-6. URL http://link.springer.com.libproxy1.nus.edu.sg/chapter/10.1007/3-540-45014-9_1.

Sašo Džeroski, Panče Panov, and Bernard Ženko. *Machine Learning, Ensemble Methods in*. Springer, 2009. URL http://link.springer.com/10.1007/978-0-387-30440-3_315.

Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *International conference on artificial intelligence and statistics*, pages 249–256, 2010. URL http://machinelearning.wustl.edu/mlpapers/paper_files/AISTATS2010_GlorotB10.pdf.

Ian J. Goodfellow, Mehdi Mirza, Da Xiao, Aaron Courville, and Yoshua Bengio. An Empirical Investigation of Catastrophic Forgeting in Gradient-Based Neural Networks. *arXiv:1312.6211 [cs, stat]*, December 2013. URL http://arxiv.org/abs/1312.6211. arXiv: 1312.6211.

Simon Haykin. *Neural Networks: A Comprehensive Foundation*. Macmillan Coll Div, New York : Toronto : New York, January 1994. ISBN 9780023527616.

Geoffrey E. Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. In *NIPS 2014 Deep Learning Workshop*, 2014. URL https://fb56552f-a-62cb3a1a-s-sites.googlegroups.com/site/deeplearningworkshopnips2014/65.pdf.

Robert A. Jacobs. Methods for combining experts' probability assessments. *Neural computation*, 7(5):867–888, 1995. URL http://www.mitpressjournals.org/doi/abs/10.1162/neco.1995.7.5.867.

Robert A. Jacobs, Michael I. Jordan, Steven J. Nowlan, and Geoffrey E. Hinton. Adaptive mixtures of local experts. *Neural computation*, 3(1):79–87, 1991. URL http://www.mitpressjournals.org/doi/abs/10.1162/neco.1991.3.1.79.

Lu Jiang, Deyu Meng, Shoou-I. Yu, Zhenzhong Lan, Shiguang Shan, and Alexander Hauptmann. Self-Paced Learning with Diversity. In *Advances in Neural Information Processing Systems*, pages 2078–2086, 2014. URL http://papers.nips.cc/paper/5648-self-paced-learning-with-diversity.pdf.

Michael I. Jordan and Robert A. Jacobs. Hierarchies of adaptive experts. In *Advances in neural information processing systems*, pages 985–992, 1992. URL http://papers.nips.cc/paper/514-hierarchies-of-adaptive-experts.

Alex Krizhevsky and Geoffrey Hinton. Learning multiple layers of features from tiny images. *Computer Science Department, University of Toronto, Tech. Rep*, 1(4):7, 2009. URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.222.9220&rep=rep1&type=pdf.

M. Pawan Kumar, Benjamin Packer, and Daphne Koller. Self-paced learning for latent variable models. In *Advances in Neural Information Processing Systems*, pages 1189–1197, 2010. URL http://papers.nips.cc/paper/3923-self-paced-learning-for-latent-variable-models.

Hugo Larochelle, Yoshua Bengio, Jérôme Louradour, and Pascal Lamblin. Exploring strategies for training deep neural networks. *The Journal of Machine Learning Research*, 10:1–40, 2009. URL http://dl.acm.org/citation.cfm?id=1577070.

Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11): 2278–2324, 1998a. URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=726791.

Yann LeCun, Cortes Corinna, and C. J. C. Burges. MNIST handwritten digit database, Yann LeCun, Corinna Cortes and Chris Burges, 1998b. URL http://yann.lecun.com/exdb/mnist/.

Sinno Jialin Pan and Qiang Yang. A Survey on Transfer Learning. *IEEE Transactions on Knowledge and Data Engineering*, 22(10):1345–1359, October 2010. ISSN 1041-4347. doi: 10.1109/TKDE.2009.191. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5288526.

Carl Edward Rasmussen. *Gaussian processes for machine learning*. Adaptive computation and machine learning. MIT Press, Cambridge, Mass, 2006. ISBN 026218253X.

Adriana Romero, Nicolas Ballas, Samira Ebrahimi Kahou, Antoine Chassang, Carlo Gatta, and Yoshua Bengio. FitNets: Hints for Thin Deep Nets. *arXiv:1412.6550 [cs]*, December 2014. URL http://arxiv.org/abs/1412.6550. arXiv: 1412.6550.

Simone Scardapane. 50 Years of Deep Learning and Beyond: an Interview with Jürgen Schmidhuber – INNS Big Data Conference Blog, 2015. URL https://innsbigdata.wordpress.com/2015/02/09/interview-with-juergen-schmidhuber/.

Jasper Snoek, Hugo Larochelle, and Ryan P. Adams. Practical Bayesian optimization of machine learning algorithms. In *Advances in Neural Information Processing Systems*, pages 2951–2959, 2012. URL http://papers.nips.cc/paper/4522-practical-bayesian-optimization-of-machine-learning-algorithms.