ECE 595 Machine Learning II

Homework 2 : Regularization of Deep Forward Networks

Murali Krishnan Rajasekharan Pillai

September 2019

Question 1 (Definition)

a) For machine learning algorithms, we define *regularization* as any modifications we make to the learning algorithm that is intended to reduce its generalizing error, but not its training error.

b) There are many reasons why we need regularization in machine learning:

   (a) The model needs extra constraints on the parameter values. This improves the performance of the model on test data.

   (b) The model needs to encode certain specific kinds of prior knowledge.

   (c) Sometimes the model learned should be generic. In other words to promote generalization.

   (d) Sometimes the problem is an under-determined problem. Regularization helps make it a determined problem.

   (e) Regularization methods, like ensemble methods, combine multiple hypothesis that explain the training data.

c) One regularization method not mentioned in Chapter 7 is **Pooling** in Convolutional Neural Networks. By using *pooling*, the output is invariant to slight spatial distortions of the input (slight changes of the location of (deep) features. Features that are sensitive to such distortions can be discarded.

Question 2 (L^2 Parameter regularization)

This is also called the **weight decay**. The objective function of a model with no bias terms can be written as:

$$\tilde{J}(\mathbf{w}; \mathbf{X}, \mathbf{y}) = \frac{\alpha}{2}\mathbf{w^T w} + J(\mathbf{w}; \mathbf{X}, \mathbf{y})$$

where $J(.)$ is te standard objective function, $\tilde{J}(.)$ is the regularized objective function, $\alpha \in [0, \infty)$ is a hyperparameter that weights the relative contribution of the norm penalty, and $\mathbf{X}$ and $\mathbf{y}$ are the model inputs and outputs. The gradient of $\tilde{J}(\mathbf{w}; \mathbf{X}, \mathbf{y})$ w.r.t. $\mathbf{w}$ is:

$$\nabla_w \tilde{J}(w; X, y) = \alpha w + \nabla_w J(w; X, y)$$

where a single gradient step to update the weights - with the learning rate ϵ - can be performed by:

$$w \leftarrow (1 - \epsilon\alpha)w - \epsilon\nabla_w J(w; X, y)$$

a) Let's make a quadratic approximation of the objective function in the neighborhood of the value of weights that obtains minimal un-regularized training cost, $w^* = \min_w J(w)$. Then the quadratic approximation $\hat{J}$ is given by:

$$\hat{J}(\theta) = J(w^*) + \frac{1}{2}(w - w^*)^T H(w - w^*)$$

where H is the Hessian matrix of J w.r.t. w evaluated at w^*. Since this is defined near the minima, $H \succeq 0$. The minimum of $\hat{J}$ occurs where its gradient

$$\nabla_w \hat{J}(w) = H(w - w^*)$$

is equal to 0.
To study the effect of weight decay, we add the weight decay gradient. We can now solve for the minimum of the regularized version of $\hat{J}$. We use the variable $\tilde{w}$ to represent the location of the minimum:

$$\begin{aligned} \alpha\tilde{w} + H(\tilde{w} - w^*) &= 0 \\ (H + \alpha I)\tilde{w} &= Hw^* \\ \tilde{w} &= (H + \alpha I)^{-1} Hw^* \end{aligned}$$

As H is real and symmetric, we can decompose it into a diagonal matrix Λ and an orthonormal basis of eigenvectors, Q, such that $H = Q\Lambda Q^T$. Applying to this decomposition, we obtain:

$$\begin{aligned} \tilde{w} &= (Q\Lambda Q^T + \alpha I)^{-1} Q\Lambda Q^T w^* \\ &= \left[Q(\Lambda + \alpha I)Q^T\right]^{-1} Q\Lambda Q^T w^* \\ &= Q(\Lambda + \alpha I)^{-1}\Lambda Q^T w^* \end{aligned}$$

We see that the effect of weight decay is to rescale w^* along the axes defined by the eigenvectors of H. Specifically, the component of w^* that is aligned with the i-th eigenvector of H is rescaled by a factor of $\frac{\lambda_i}{\lambda_i + \alpha}$

b) Let's say that the eigenvalues of $\boldsymbol{H}$ are $\lambda_i = \{\lambda_1, \lambda_2, \cdots, \lambda_n\}$. There exists two extreme cases,

- When $\lambda_i \gg \alpha \implies \frac{\lambda_i}{\lambda_i + \alpha} \approx 1$. In this case the effect of regularization is relatively small. Hence the parameters along direction i i preserved relatively intact,
- The next possible scenario is when: $\lambda_i \ll \alpha \implies \frac{\lambda_i}{\lambda_i + \alpha} \approx \frac{1}{\alpha}$. Here, the components will be shrunk to nearly zero magnitude by the regularization. In directions that do not contribute to reducing the objective function, a small eigenvalue of the Hessian tells us that movement in this direction will not significantly increase the gradient. Components of the weight vector corresponding to such unimportant directions are decayed away through the use of the regularization throughout training.

Both these scenarios are observed when the condition number of $\boldsymbol{H}$ is high (which means that there is a large difference between the maximum and minimum eigenvalues of the $\boldsymbol{H}$). Hence we can assume that the dimensionality of the matrix used to invert during trainig is small and hence requires lesser time for training. If $\boldsymbol{H}$ has a small condition number, then all eigenvalues are of comparable values. This means that the L^2 regularizer has more parameters to regularize. This might increase the dimensionality of the matrix to be inverted for the training process, which in-turn increases the training time.

Question 3 (L^1 Regularization)

a) Formally, L^1 regularization on the model parameter $\mathbf{w}$ is defined as

$$\Omega(\boldsymbol{\theta}) = \|\mathbf{w}\|_1 = \sum_i |w_i|$$

The L^1 regularized objective function $\tilde{J}(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y})$ is given by:

$$\tilde{J}(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}) = \alpha \|\boldsymbol{w}\|_1 + J(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y})$$

with the corresponding gradient is,

$$\nabla_{\boldsymbol{w}} \tilde{J}(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}) = \alpha \text{ sign}(\boldsymbol{w}) + \nabla_{\boldsymbol{w}} J(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y})$$

where $\text{sign}(\boldsymbol{w})$ is simply the sign of $\boldsymbol{w}$ applied element-wise. The cost function can be approximated around the minima ($\boldsymbol{w^*}$) as,

$$\hat{J}(\theta) = J(w^*) + \frac{1}{2}(\boldsymbol{w} - \boldsymbol{w^*})^T \boldsymbol{H} (\boldsymbol{w} - \boldsymbol{w^*}) \tag{1}$$

The gradient of this approximation near the minima is:

$$\nabla_{\boldsymbol{w}} \hat{J}(\boldsymbol{w}) = \boldsymbol{H}(\boldsymbol{w} - \boldsymbol{w^*})$$

where $\boldsymbol{H}$ is the Hessian matrix of J w.r.t. $\boldsymbol{w}$ evaluated at w^*. If there exists no correlation between the input features the Hessian of the objective function would be a diagonal matrix. (In practice this can be achieved through Principal Component Analysis on the input features. Therefore,

$$\boldsymbol{H} = \text{diag}([H_{1,1}, \cdots H_{n,n}])$$

Hence, (**??**) can be written as:

$$\hat{J}(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y} = J(\boldsymbol{w^*}; \boldsymbol{X}, \boldsymbol{y}) + \sum_i \left[\frac{1}{2} H_{i,i} (\boldsymbol{w}_i - \boldsymbol{w}_i^*)^2 + \alpha |w_i| \right]$$

$$\implies \frac{d(\hat{J}(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y})}{dw_i} = H_{i,i} w_i + \alpha \text{ sign}(w_i) - H_{i,i} w_i^*$$

$$\implies \frac{d(\hat{J}(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y})}{dw_i} = 0 \qquad (\because \text{ at minima, if possible})$$

$$\implies H_{i,i} w_i + \alpha \text{ sign}(w_i) x = H_{i,i} w_i^*$$

$$\implies w_i = w_i^* - \frac{\alpha}{H_{i,i}} \text{ sign}(w_i) \qquad \text{(Equation (a))}$$

If Equation (a) is true, then $\text{sign}(w_i^*) = \text{sign}(w_i)$

$$w_i = w_i^* - \frac{\alpha}{H_{i,i}} \text{sign}(w_i^*)$$

$$w_i = \text{sign}(w_i^*) \left[|w_i^*| - \frac{\alpha}{H_{i,i}} \right]$$

If Equation (a) is not possible, The problem of minimizing this approximate cost function has an analytical solutions (for each dimension i), with the following form:

$$w_i = \text{sign}(w_i^*) \max \left\{ |w_i^*| - \frac{\alpha}{H_{i,i}} \right\}$$

b) By inspecting the gradient, we can immediately see that the effect of L^1 regularization is quite different from that of L^2 regularization. We can see that the regularization contribution to the gradient no longer scales linearly with each w_i; instead it is constant factor with a sign equal to $\text{sign}(w_i)$. This promotes sparsity in the model, and this property has been extensively used as a **feature selection mechanism**. The L^1 penalty causes a subset of the weights to become zero, suggesting that the corresponding features may safely be discarded. (***Interesting note***: L^1 regularization can be thought of us putting a isotropic Lapace prior over the weights of the model. $\log p(w) = \sum_i \log \text{Laplace}\left(w_i; 0, \frac{1}{\alpha}\right) = -\alpha \left\|w\right\|_1 + n \log \alpha - n \log 2$)

Question 4 (Constrained Optimization)

a) Penalties alone can cause non-convex optimization procedures to gets stuck in local minimal corresponding to small θ. In the case of neural-networks, this manifests as a a model that trains with "dead units". These units do not contribute much to the behavior of the function learned by the netowrk because the weights going into or out of them are all very small. When training with norm penalties, these configurations can be locally optimal, even if it is possible to significantly reduce J by making the weights larger.

b) Using explicit constraints with re-projections prevent producing local optima, because they do not encourage weights to approach the origin (aka 0). Explicit constraints implemented by re-projections have an effect only when the weights become large and attempt to leave the constraint region. Explicit constraints with re-projection can be useful because they impose some stability on the optimization procedure. When using high learning rates, it is possible to encounter a positive feedback loop in which large weights induce large gradients, which then induce a large update to the weights. If these updates consistently increase the size of the weights, then θ rapidly moves away from the origin until numerical overflow occurs. Explicit constraints with re-projection prevent this feedback loop from continuing to increase the magnitude of the weights without bound.

c) ***Hinton et al. (2012)*** recommend the strategy of constraining the norm of each *column* of the weight matrix of a neural net layer, rather than constraining the Frobenius norm of the entire weight matrix. Constraining the norm of each column separately prevents any one hidden unit from having very large weights. This constraint would be similar to the L^2 weight decay but with a separate KKT multiplier for the weights of each hidden unit. Each of these KKT conditions would be dynamically updated separately to make each hidden unit obey the constraint. Since this method is usually implemented as an explicit constraint with re-projection, this makes sure that the regularization effect only kicks in for each weight after a minimum threshold which is defined by the explicit constraint.

Question 5 (Regularization & Under-Constrained Problems)

There are two kinds of problem in machine learning which depends on the method of solving, namely those which have **closed-form solutions** and those which are solved using an **iterative algorithm**.

For the problems which have a closed-form solution, given the input matrix $\boldsymbol{X}$, solving these involve inverting the matrix $\boldsymbol{X^T X}$. This is not possible when $\boldsymbol{X^T X}$ is singular (*in other words it is an under-constrained system*). $\boldsymbol{X^T X}$ is singular whenever the data-generating distribution truly has no variance in some direction, or when no variance is observed in some direction because there are fewer examples (rows of $\boldsymbol{X}$) than input features (columns of $\boldsymbol{X}$). Regularization leads to inverting the matrix $\boldsymbol{X^T X} + \alpha \boldsymbol{I}$ instead. This regularized matrix is guaranteed to be invertible.

For the problems which do not have a closed form solution, iterative procedures like stochastic gradient descent will, *in theory*, not halt. In practice, SGD will reach sufficiently large weights to cause numerical overflow. Most forms of regularization can guarantee the convergence of iterative methods applied to undetermined problems. For example, weight decay will cause gradient descent to quit increasing the magnitude of the weights when the slope of the likelihood is equal to the weight decay coefficient. This means that the regularization constrains the model to choose a more simpler model than going for more complicated model (which is in conformance with *Occam's razor*).

Question 6 (Dataset Augmentation & Noise Robustness)

a) In the case of image object recognition, the classifier needs to take a complicated, high-dimensional input $\boldsymbol{x}$ and summarize it with a a single category identity y. This means that the main task facing a classifier is to be invariant to a wide variety of transformations. Some of the operations that can be performed on images as regularization techniques are as follows:

   i) Translating the training images a few images in each direction.

   ii) Rotating the image (such that it does not change the image class)

   iii) Scaling the image

   iv) Injecting noise into the input of a neural network

b) Data augmentation has to be done in such a way that the class of the object is not changed via augmenting the input. For example, OCR tasks require recognizing the difference between "b" and "d", and the difference between "6" and "9". So, horizontal flips and 180° rotations are not appropriate ways of augmenting datasets for these tasks.

c) Let's consider the LLS cost associated with training a function $\hat{y}(\boldsymbol{x})$ that maps a set of features $\boldsymbol{x}$ to a sclar y. The cost between the model's predictions, $\hat{y}(x)$, and the true values, y is given by:

$$J = \mathbb{E}_{p(x,y)}[(\hat{y}(x) - y)^2]$$

Let's include a random perturbation $\epsilon_{\boldsymbol{w}} \sim \mathcal{N}(\mathbf{0}, \eta \boldsymbol{I})$ to the network weights. Let's say that the corresponding prediction with the random perturbation is $\hat{y}_{\epsilon_w}(x)$ Now we have,

$$\begin{aligned}
\tilde{J}_{\epsilon_w} &= \mathbb{E}_{p(x,y,\epsilon_w)}\left[(\hat{y}_{\epsilon_w}(x) - y(x))^2\right] \\
&= \mathbb{E}_{p(x,y,\epsilon_w)}\left[(\hat{y}_{\epsilon_w}(x)^2 - 2\hat{y}_{\epsilon_w}(x)y(x) + y(x)^2\right] \\
&= \mathbb{E}_{p(x,y,\epsilon_w)}\left[(\hat{y}(x) + \epsilon_w x)^2 - 2(\hat{y}(x) + \epsilon_w w)y(x) + y(x)^2\right] && (\because \hat{y}_{\epsilon_w} = \hat{y} + \epsilon_w x) \\
&= \mathbb{E}_{p(x,y,\epsilon_w)}\left[\hat{y}(x)^2 + (\epsilon_w x)^2 + 2\hat{y}(x)\epsilon_w x - 2\hat{y}(x)y(x) - \epsilon_w w y(x) + y(x)^2\right] \\
&= \mathbb{E}_{p(x,y,\epsilon_w)}\left[\hat{y}(x)^2 - 2\hat{y}(x)y(x) + y(x)^2\right] + \mathbb{E}_{p(x,y,\epsilon_w)}\left[(\epsilon_w x)^2 + 2\hat{y}(x)\epsilon_w x - \epsilon_w w y(x)\right] \\
&= \mathbb{E}_{p(x,y,\epsilon_w)}\left[(\hat{y}(x) - y(x))^2\right] + \mathbb{E}_{p(x,y,\epsilon_w)}\left[\epsilon_w^2\right] \cdot \mathbb{E}_{p(x,y,\epsilon_w)}\left[x^2\right] \\
&= \mathbb{E}_{p(x,y,\epsilon_w)}\left[(\hat{y}(x) - y(x))^2\right] + \boldsymbol{\eta}\mathbb{E}_{\boldsymbol{p(x,y,\epsilon_w)}}\left[\|\boldsymbol{\nabla_w \hat{y}(x)}\|_{\boldsymbol{2}}^{\boldsymbol{2}}\right]
\end{aligned}$$

As we can see here, by adding the noise to the weights, we *essentially* add a **regularization term** to the objective function where no noise is added to the weights. This discourages a prediction that is sensitive to small changes in $\boldsymbol{w}$.

d) *Label smoothing* is a regularization method used to make the model robust to uncertainties in the labels for the dataset. The straight-forward way to achieve this is by assuming a small constant probability ϵ, the training set label y is correct with probability $1 - \epsilon$, and otherwise any of the other possible labels might be correct. This assumptions can be incorporated into the cost function analytically, rather than by explicitly drawing noise samples. For example, **label smoothing** regularizes a model based on a softmax with k output values by replacing the hard 0 and 1 classification targets with targets of $\frac{\epsilon}{k-1}$ and $1-\epsilon$, respectively. Label smoothing has the advantage of preventing the pursuit of hard probabilities without discouraging correct classification. The softmax function is defined as

$$\sigma(\boldsymbol{z}_i) = \frac{e^{\boldsymbol{z}_i}}{\sum_{j=1}^{K} e^{\boldsymbol{z}_j}} \text{ for } i = 1, \cdots, K \text{ and } \boldsymbol{z} = (z_1, \cdots, z_K) \in \mathbb{R}^K$$

The softmax function essentially represents $p_{\text{model}}(\boldsymbol{y}|\boldsymbol{x}, \boldsymbol{\theta})$. The softmax-function with the cross-entropy loss basically incorporates the model of label noise, rather than sampling independently.

Question 7 (Generative Training)

a) **Generative training** explicitly models the actual distribution of each class. This model learns the joint probability distribution $p(x, y)$, where x is the input features and y is the associated label. It predicts the conditional probability with the help of Bayes Theorem.

Discriminative training models the decision boundary between classes. This model learns the conditional probability distribution $p(y|x)$.

Let's make this definition a bit more formal: Training a classifier involves estimating $f : \mathcal{X} \to \mathcal{Y}$ or $P(Y|X)$ In the generative models,

- Assume some functional form for $\boldsymbol{P(y), P(X|y)}$
- Estimate parameters for $\boldsymbol{P(y), P(X|y)}$ from the training data
- Use Bayes rule to calclate $\boldsymbol{P(y|X)}$
- eg: Naive Bayes, Bayesian Networks, Markov Random Fields, Hidden Markov models

In the discriminative models,

- Assume some functional form for $\boldsymbol{P(y|X)}$
- Estimate parameters for $\boldsymbol{P(y|X)}$ from the training data

- eg: Logistic Regression, SVMs, Neural Networks

However, at the end of the day both models predict the conditional probability p(label|data), but both models learn different probabilities.

b) i) In **Semi-Supervised Learning (SSL)** both unlabeled examples from $P(x)$ and labeled examples from $P(x, y)$ are used to estimate $P(y|x)$, or predict $\boldsymbol{y}$ from $\boldsymbol{x}$. Typically, in semi-supervised learning we provide much more unlabelled data than labelled data. It has a generative component (the unsupervised learning) which learns a representation such that similar inputs have similar representations in the representations space. The discriminative component, then uses these representations to build the classification boundaries between the classes. However, SSL can be achieved by constructing models where generative model of either $P(\boldsymbol{x})$ or $P(\boldsymbol{x}, \boldsymbol{y})$ shares parameters with a discriminative model of $P(\boldsymbol{y}|\boldsymbol{x})$. Then, the supervised criteria $-\log P(y|x)$ can be traded of with the unsupervised or generative criteria $-\log P(x)$ or $-\log P(x, y)$. The generative criteria then expresses a particular form of prio belief about the solution to the supervised learning problem, namely the structure of $P(x)$ is connected to the structure of $P(y|x)$ in away that is captured by the shared parameterization.

ii) **Multi-task Learning** is a way to improve the generalization by pooling the examples (which can be seen as soft constraints imposed on the parameters) arising out of several tasks.

The above figure illustrates a very common form of multi-task learning, in which different supervised tasks (predicting $\boldsymbol{y}^{(i)}$ share the same input $\boldsymbol{x}$, as well as some intermediate-level representation $\boldsymbol{h}^{(shared)}$, capturing a common pool of factors. Improved generalization and generalization error bounds can be achieved because of the shared parameters, for which statistical strength can be greatly improved. However, this is only valid if some assumptions about the statistical relationship between the different tasks are valid, meaning that there is something shared across some of the tasks.

The intermediate hidden layers, (represented by $h^{(\text{shared})}$) is essentially a generative task as there is no labelled data for the model to learn from. The later layers which correspond to specific tasks are basically discriminative part of the model which learns a discriminative function to distinguish between the various tasks.

iii) A popular way to regularize a model is to use constraints, particularly by forcing sets of parameters to be equal. This method of regularization is often referred to as **parameter sharing**, because we interpret the various models or model components as sharing a unique set of parameters. Here the update/learning of intermediate shared layers, (represented by $h^{(\text{shared})}$) is essentially a generative task as there is no labelled data for the model to learn from. The upper layers of the Neural network which correspond to specific tasks are basically the discriminative part of the model which learns a discriminative function to distinguish between the various tasks.

Question 8 (Early Stopping)

a) A validation set is one of the two disjoint sets created by splitting the training data. The other subset is used to learn the parameters. The validation set is is used to estimate the generalization error during or after training, allowing for hyper-parameters to be updated accordingly. In simple words, validation set is used to guide the selection of the hyper-parameters which cannot be tuned during the training operation. Since the validation set is used to "train" the hyper-parameters, the validation set error will underestimate the generalization error.

The test set is is composed of examples coming from the same distribution as the training set and is used to estimate the generalization error of a learner, after the learning process has completed. This dataset cannot

be used to make any choices about the model, including the model's hyper-parameters. For this reason, no example of the test set can be used in the validation set.

b) When training large models with sufficient representation capacity to overfit the task, the training error decreases over time, but validation set error begins to rise again. This means that we can obtain a model with better validation set error (and thus *hopefully* better test set error) by returning to the parameter setting at the point in time with the lowest validation set error. Every-time the error on the validation set improves, a copy of the model parameters are stored. When the training algorithm terminates, we return these parameters rather than the latest parameters. The algorithm terminates when no parameters have improves over the best recorded validation set error for some pre-specified number of iteration. This strategy is known as **early stopping**. It is a very efficient hyper-parameter selection algorithm. Through early-stopping the effective capacity of the model is controlled by determining how many steps it can take to fit the training set.

c) The challenge in **re-training** with the added dataset is that there is no good way of knowing whether to retrain for the same number of parameter updates or the same number of passes through the dataset. This is because, in the second round of training, each pass through the dataset will require more parameter updates as the training set is bigger, hence it is unsure if the number of parameter updates is valid anymore for the larger dataset. A common method people use is by having the same number of passes over the larger dataset.

d) The challenge in **resuming training** rather than re-training is that, there no longer exists a guide for when to stop in terms of a number of steps. One way to overcome this challenge is to monitor the average loss function on the validation set and continue training until it falls below the value of the training set objective at which the early stopping procedure halted. This strategy avoids the high cost of retraining the model from the scratch, but is not as well-behaved.

Question 9 (Early Stopping as an L^2 Regularizer)

To compare Early-stopping with the classical L^2 regularization, let's examine the setting where the only parameters are linear weights. In the neighborhood of the empirically optimal value of the weights $\boldsymbol{w}^*$, we can model the cost function J with a quadratic approximation as:

$$\hat{J}(\boldsymbol{\theta}) = J(\boldsymbol{w}^*) + \frac{1}{2}(\boldsymbol{w} - \boldsymbol{w}^*)^T \boldsymbol{H}(\boldsymbol{w} - \boldsymbol{w}^*)$$

where $\boldsymbol{H}$ is the Hessian matrix of J w.r.t to $\boldsymbol{w}$ evaluated at $\boldsymbol{w}^*$. Given the assumption that $\boldsymbol{w}^*$ is a minimum of $J(\boldsymbol{w})$, $\boldsymbol{H}$ is positive semi-definite.

Under a local Taylor series approximation, the gradient is given by

$$\nabla_{\boldsymbol{w}} \hat{J}(\boldsymbol{w}) = \boldsymbol{H}(\boldsymbol{w} - \boldsymbol{w}^*)$$

Following the trajectory followed by the parameter vector during training. Let's assume that the initial parameter vector to the origin $\boldsymbol{w}^{(0)} = \mathbf{0}$. Let us study the approximate behavior of the gradient descent on J by analyzing gradient descent on $\hat{J}$:

$$\begin{aligned}\boldsymbol{w}^{(\tau)} &= \boldsymbol{w}^{(\tau-1)} - \epsilon \nabla_{\boldsymbol{w}} \hat{J}(\boldsymbol{w}^{(\tau-1)}) \\ &= \boldsymbol{w}^{(\tau-1)} - \epsilon \boldsymbol{H}(\boldsymbol{w}^{(\tau-1)} - \boldsymbol{w}^{*}) \\ \boldsymbol{w}^{\tau} - \boldsymbol{w}^{*} &= (\boldsymbol{I} - \epsilon \boldsymbol{H})(\boldsymbol{w}^{(\tau-1)} - \boldsymbol{w}^{*})\end{aligned}$$

Re-writing the above equations in the space of the eigenvectors of $\boldsymbol{H}$, exploiting the eigen-decomposition of $\boldsymbol{H}$: $\boldsymbol{H} = \boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^{T}$, where $\boldsymbol{\Lambda}$ is a diagonal matrix and $\boldsymbol{Q}$ is an orthonormal basis of eigenvectors.

$$\begin{aligned}\boldsymbol{w}^{(\tau)} - \boldsymbol{w}^{*} &= (\boldsymbol{Q}\boldsymbol{Q}^{T} - \epsilon \boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^{T})(\boldsymbol{w}^{(\tau-1)} - \boldsymbol{w}^{*}) \\ \boldsymbol{Q}^{T}(\boldsymbol{w}^{(\tau)} - \boldsymbol{w}^{*}) &= (\boldsymbol{I} - \epsilon \boldsymbol{\Lambda})\boldsymbol{Q}^{T}(\boldsymbol{w}^{(\tau-1)} - \boldsymbol{w}^{*})\end{aligned}$$

Assuming that $\boldsymbol{w}^{(0)} = 0$ and that ϵ is chosen to be small enough to guarantee $|1 - \epsilon\lambda_i| < 1$, the parameter trajectory during training after τ parameter updates is:

$$\boldsymbol{Q}^{T}\boldsymbol{w}^{(\tau)} = [\boldsymbol{I} - (\boldsymbol{I} - \epsilon\Lambda)^{\tau}]\boldsymbol{Q}^{T}\boldsymbol{w}^{*} \tag{2}$$

The expression for $\boldsymbol{Q}^{T}\tilde{\boldsymbol{w}}$ for L^2 regularization can be re-arranged as:

$$\begin{aligned}\boldsymbol{Q}^{T}\tilde{\boldsymbol{w}} &= (\Lambda + \alpha\boldsymbol{I})^{-1}\boldsymbol{\Lambda}\boldsymbol{Q}^{T}\boldsymbol{w}^{*} \\ \boldsymbol{Q}^{T}\tilde{\boldsymbol{w}} &= (\Lambda + \alpha\boldsymbol{I})^{-1}[(\boldsymbol{\Lambda} + \alpha\boldsymbol{I}) - \alpha\boldsymbol{I}]\boldsymbol{Q}^{T}\boldsymbol{w}^{*} \\ \boldsymbol{Q}^{T}\tilde{\boldsymbol{w}} &= [\boldsymbol{I} - \alpha(\Lambda + \alpha\boldsymbol{I})^{-1}]\boldsymbol{Q}^{T}\boldsymbol{w}^{*} \qquad (3) \\ & \qquad (4)\end{aligned}$$

Comparing equation (**??**) & (**??**), if we choose ϵ, α and τ such that:

$$(\boldsymbol{I} - \epsilon\Lambda)^{\tau} = (\Lambda + \alpha\boldsymbol{I})^{-1}\alpha \tag{5}$$

then L^2 regularization and early stopping can be seen as equivalent (under the quadratic approximation of the objective function). Going further, for each row i in (**??**), we can write

$$\begin{aligned}(1 - \epsilon\lambda_i)^{\tau} &= (\lambda_i + \alpha)^{-1}\alpha \\ \tau \log(1 - \epsilon\lambda_i) &= (-1)\log(\frac{\lambda_i}{\alpha} + 1) \\ \tau(\epsilon\lambda_i) &\approx (-1)\frac{-\lambda_i}{\alpha} \qquad (\because \log(1 - x) \approx x) \\ \alpha &\approx \frac{1}{\epsilon\tau}\end{aligned}$$

Under these assumptions, the number of training iterations τ plays a role inversely proportional to the L^2 regularization parameter, and the inverse of $\tau\epsilon$ plays the role of the weight decay coefficient.

Question 10 (Bootstrap Aggregating (Bagging) and Boosting)

a) **Bagging** (short for **bootstrap aggregating**) is a technique for reducing the generalization error by combining several models. The idea is to train several different models separately, then have all the models vote on the output for test examples. Bagging allows the use of same kind of model, training algorithm and objective function to be reused several times. Specifically, bagging involves constructing k different datasets. Each dataset has the same number of examples as the original dataset, but each dataset is constructed by sampling with replacement from the original dataset. This means that, with high probability, each dataset is missing some of the examples from the original dataset and contains several duplicate examples. The differences between which examples are included in each dataset result in differences between the trained models.

Since the representations learned by different models depend on the difference in the datasets, it is not possible to make *baggging* useful when the entire training set is used to train all the models. In this case the various models learn very similar representations of the data, that the advantage of bagging is missed.

b) Boosting refers to the technique of constructing ensembles with higher capacity than the individual models. Boosting has been applied in interpreting an individual neural network as an ensemble by incrementally adding hidden units to the network.

Question 11 (Dropout)

a) ***Dropout*** was introduced as a computationally inexpensive but powerful method of regularizing a broad family of models. It provides an inexpensive approximation to training and evaluating a bagged ensemble of exponentially many neural networks. In most modern neural networks, based on a series of affine transformations and nonlinearities, we can effectively remove a unit from a network by multiplying its output value by zero. This results in a set of possible sub-networks from the original neural network (an ensemble of subnetworks). Dropout trains the ensemble consisting of all subnetworks that can be formed by removing non-output units from an underlying base network by applying these random **binary masks**. In practice, each time an example is loaded into a mini-batch, a random binary mask is applied to all input and hidden units in the network. This creates an ensemble of the subnetworks, on which the forward propagation, back-propagation and the learning update is performed as usual.

b) In **Dropout**, the ensemble of sub-network models share parameters, with each model inheriting a different subset of parameters from the parent neural network. This parameter sharing makes it possible to represent an exponential number of models with a tractable amount of memory. Additionally, most models are not trained explicitly due to the size of parent network. Instead, a tiny fraction of the possible subnetworks are each trained for a single step, and the parameter sharing causes the remaining subnetworks to arrive at good settings of the parameters.

In **bagging**, k different models are defined and k different datasets are created by sampling from the training set with replacement. Then the i^{th} model is trained on the i^{th} dataset. These models are independently defined with no parameter sharing. Contrary to *dropout*, each model is also trained to convergence on it's respective training dataset. Due to the definition of the independent models, large number of subnetworks cannot be "learned" because it is not possible to achieve in a tractable amount of memory. Finally to make a prediction the *bagged ensemble* should accumulate votes from all its members.

Question 12 (Dropout Continued)

a) In dropout, each sub-model defined by a mask vector $\boldsymbol{\mu}$ defines a probability distribution $p(y|x, \boldsymbol{\mu})$. The geometric mean of the ensemble members' predicted distributions can provide a good approximation to the predictions of the ensemble (*in lieu* of arithmetic mean which can become intractable in large networks). To guarantee that the geometric mean is a valid probability distribution, the sub-models cannot assign a 0 probability to any event. The un-normalized probability distribution defined directly by the geometric mean is given by

$$\tilde{p}_{\text{ensemble}}(y|\boldsymbol{x}) = \sqrt[2^d]{\prod_{\boldsymbol{\mu}} p(y|\boldsymbol{x}, \boldsymbol{\mu})},$$

where d is the number of units that may be dropped. To make the predictions, the probabilities should be renormalized:

$$p_{\text{ensemble}}(y|\boldsymbol{x}) = \frac{\tilde{p}_{\text{ensemble}}(y|\boldsymbol{x})}{\sum_{y'} \tilde{p}_{\text{ensemble}}(y'|\boldsymbol{x})}$$

This brings us to the ***weight scaling inference rule***, a key insight involved in dropout is that we can approxmiate p_{ensemble} by evaluating $p(y|\boldsymbol{x})$ in one model: the model with all units, but with the weights going out of unit i multiplied by the probability of including unit i. The motivation for this modification is to capture the right expected value of the output from that unit.
Goodfellow et al (2013a) found that weight scaling approximation can work better than Monte Carlo approximations to the ensemble predictor. This held true even when the MC approximation was allowed to sample upto 1,000 subnetworks. The comparison was done in terms of **classification accuracy**.

b) (a) **Dropout boosting** is an algorithm that injects exactly the same noise as DropOut. The objective function for each (sub-network, example) pair in dropout-boosting is the likelihood of the data according to the ensemble; however only the parameters of the current sub-network maybe updated for each example. Ordinary dropout performs bagging by maximizing the likelihood of the correct target for the current example under the current sub-netwrok, whereas dropout boosting takes into account the contributions of other subnetworks, in a manner reminiscent of boosting.

(b) **DropConnect** is a method which works similar to DropOut and is intended to prevent the "co-adaptation" of units in the neural network. It is very similar to DropOut and can be thought of as a more general form of DropOut. In DropConnect, we apply the binary mask to individual weights of the network rather than the individual nodes. In this case the node can remain partially active, unlike in DropOut.

(c) Instead of using a binary mask $\boldsymbol{\mu}$ (or for that matter any finite number of values for $\boldsymbol{\mu}$), *Srivastava et al. (2014)* showed that multiplying the weights by $\mu \sum \mathcal{N}(\mathbf{1}, \boldsymbol{I})$ can outperform DropOut based on binary masks. Since $\mathbb{E}[\boldsymbol{\mu}] = \mathbf{1}$, the standard network automatically implements approximate inference in the ensemble, without needing any weight scaling.