Question 1. Getting Started

You may typeset your homework in latex or submit neatly handwritten and scanned solutions. Please make sure to start each question on a new page, as grading (with Gradescope) is much easier that way! Deliverables:

1. Submit a PDF of your writeup to assignment on Gradescope, “HW[n] Write-Up”
2. Submit all code needed to reproduce your results, “HW[n] Code”.
3. Submit your test set evaluation results, “HW[n] Test Set”.

After you’ve submitted your homework, be sure to watch out for the self-grade form.

(a) Who else did you work with on this homework? In case of course events, just describe the group. How did you work on this homework? Any comments about the homework?

I worked with Weiran Liu and Katherine Li. It’s a pleasant to see suck a long homework at the end of the semester. Struggling with the project and a long homework, I don’t have much more to say. Making long homework just makes people frustrated. If someone reads this, it means I didn’t drop it.

(b) Please copy the following statement and sign next to it. We just want to make it *extra* clear so that no one inadvertently cheats.

I certify that all solutions are entirely in my words and that I have not looked at another student’s solutions. I have credited all external sources in this write up.

I certify that all solutions are entirely in my words and that I have not looked at another student’s solutions. I have credited all external sources in this write up.

Signature: 盛焕杰

Question 2. Gradient boosting and early stopping

In this problem we show how the greedy algorithms Matching Pursuit (a simplified version of the Orthogonal Matching Pursuit that you might have seen in EE16A) and AdaBoost which you have seen in class, can both be interpreted as taking steps greedily in a direction which is closest to the negative gradient in a restricted space. We start by connecting gradient descent on the square loss with Matching Pursuit.

Gradient descent on square loss So far in lecture, we have considered the squared error loss $L(\vec{w}) := \frac{1}{2}\|\vec{y} - \mathbf{X}\vec{w}\|_2^2$ as a function of the variable $\vec{w}$. When running gradient descent we took the derivative of L with respect to $\vec{w}$ and obtained iterations

$$\vec{w}^{t+1} = \vec{w}^t - \alpha_t \mathbf{X}^\top(\mathbf{X}\vec{w} - \vec{y}) \tag{1}$$

We can substitute $\vec{v} = \mathbf{X}\vec{w} \in \mathbb{R}^n$ and equivalently consider the loss function as a map $\mathcal{L} : \mathcal{V} \mapsto \mathbb{R}$ mapping from the Euclidean space $\mathcal{V} = \mathbb{R}^n$ to real values:

$$\mathcal{L}(\vec{v}) = \frac{1}{2}\|\vec{y} - \vec{v}\|^2.$$

In order to minimize this new loss $\mathcal{L}$ with respect to $\vec{v}$, we could imagine simply running gradient descent on this loss as follows:

$$\vec{v}^{t+1} = \vec{v}^t - \alpha_t \nabla_{\vec{v}} \mathcal{L}(\vec{v})|_{\vec{v}=\vec{v}^t}, \tag{2}$$

for some suitably chosen step size $\alpha_t > 0$.

(a) **Compute the gradient $\nabla_{\vec{v}}\mathcal{L}(\vec{v})$. Write down the gradient descent update for $\vec{v}^{t+1}$. What is the gradient $\nabla_{\vec{v}}\mathcal{L}(\vec{v})$ evaluated at $\vec{v} = \mathbf{X}\vec{w}$ which we denote as $\nabla_{\vec{v}}\mathcal{L}(\vec{v})\big|_{\vec{v}=\mathbf{X}\vec{w}}$? For $\vec{v}^t = \mathbf{X}\vec{w}^t$, compute the expression for $\vec{v}^{t+1}$ using update** (2).

Now we compare this update with the one obtained by running the gradient descent update of $\vec{w}$ on the loss L. In particular, using equation (1), **write down the expression for $\mathbf{X}\vec{w}^{t+1}$. How do you interpret the difference between the updates $\mathbf{X}\vec{w}^{t+1}$ and $\vec{v}^{t+1}$?** Think in terms of underlying subspaces in $\mathbb{R}^d$ in which the updates lie.

Compute the gradient $\nabla_{\vec{v}}\mathcal{L}(\vec{v})$. Write down the gradient descent update for $\vec{v}^{t+1}$.

$$\begin{aligned}&\nabla_{\vec{v}}\mathcal{L}(\vec{v})\\ =&\frac{1}{2}\cdot 2(\vec{v} - \vec{y})\\ =&\vec{v} - \vec{y}\end{aligned}$$

$$\begin{aligned}\vec{v}^{t+1} &= \vec{v}^t - \alpha_t \nabla_{\vec{v}}\mathcal{L}(\vec{v})|_{\vec{v}=\vec{v}^t}\\ \vec{v}^{t+1} &= \vec{v}^t - \alpha_t(\vec{v}^t - \vec{y})\end{aligned}$$

What is the gradient $\nabla_{\vec{v}}\mathcal{L}(\vec{v})$ evaluated at $\vec{v} = \mathbf{X}\vec{w}$ which we denote as $\nabla_{\vec{v}}\mathcal{L}(\vec{v})\big|_{\vec{v}=\mathbf{X}\vec{w}}$? For $\vec{v}^t = \mathbf{X}\vec{w}^t$, compute the expression for $\vec{v}^{t+1}$ using update (2)

$$\nabla_{\vec{v}}\mathcal{L}(\vec{v})\big|_{\vec{v}=\mathbf{X}\vec{w}}$$

$$
\begin{aligned}
&= \vec{v} - \vec{y} \\
&= \mathbf{X}\vec{w} - \vec{y}
\end{aligned}
$$

$$
\begin{aligned}
\vec{v}^{t+1} &= \vec{v}^{t} - \alpha_t \nabla_{\vec{v}} \mathcal{L}(\vec{v})|_{\vec{v}=\vec{v}^t} \\
\vec{v}^{t+1} &= \vec{v}^{t} - \alpha_t (\vec{v}^{t} - \vec{y}) \\
\vec{v}^{t+1} &= \mathbf{X}\vec{w}^{t} - \alpha_t (\mathbf{X}\vec{w}^{t} - \vec{y})
\end{aligned}
$$

write down the expression for $\mathbf{X}\vec{w}^{t+1}$

Copy the equation of $\vec{w}^{t+1}$ from above:

$$
\vec{w}^{t+1} = \vec{w}^{t} - \alpha_t \mathbf{X}^\top (\mathbf{X}\vec{w}^{t} - \vec{y})
$$

Substitute it into $\mathbf{X}\vec{w}^{t+1}$

$$
\begin{aligned}
&\mathbf{X}\vec{w}^{t+1} \\
=&\mathbf{X}(\vec{w}^{t} - \alpha_t \mathbf{X}^\top (\mathbf{X}\vec{w}^{t} - \vec{y})) \\
=&\mathbf{X}\vec{w}^{t} - \alpha_t \mathbf{X}\mathbf{X}^\top (\mathbf{X}\vec{w}^{t} - \vec{y})
\end{aligned}
$$

How do you interpret the difference between the updates $\mathbf{X}\vec{w}^{t+1}$ and $\vec{v}^{t+1}$?

In $\mathbf{X}\vec{w}^{t+1}$, the "step size" is determined by $\alpha_t \mathbf{X}\mathbf{X}^\top$; in $\vec{v}^{t+1}$ the step size is determined by α_t.
That is $\vec{v}$ is directly updated in $\mathbb{R}^n$ space but $\mathbf{X}\vec{w}^{t+1}$ is updated in $\mathbb{R}^d$ space and then project to $\mathbb{R}^n$ space.
$\vec{v}^{t+1}$ is moving towards $\vec{y}$; $\mathbf{X}\vec{w}^{t+1}$ is moving towards the projection of $\vec{y}$ in the column space of $\mathbf{X}$ as $\mathbf{X}\vec{w}^{*}$.

(b) As we see in the previous part, the updates $\mathbf{X}\vec{w}^{t+1}$ and $\vec{v}^{t+1}$ derived by taking gradients of L and $\mathcal{L}$ respectively, are not equivalent. In general, we may not be able to run the gradient updates (2) exactly, as there might be constraints on the update we want to make. Sometimes, these constraints are consequences of the problem formulation (in the previous part for example, we wanted a linear fit for y depending on $\vec{x}$). Sometimes, there are further constraints that come from computational considerations, as in the follow-up parts dealing with AdaBoost). In this problem, we will see that if instead of (2) we take a direction closest to the gradient in some restricted space, both procedures actually match.

In the specific example of linear regression, we want to find a vector $\widetilde{\vec{v}}^t$ which can be expressed as a linear function of our data matrix $\mathbf{X}$. For this purpose we restrict our search space to the column space of $\mathbf{X}$ (a subspace of $\mathbb{R}^n$) in which all vectors $\widetilde{\vec{v}}$ can be expressed by $\widetilde{\vec{v}} = \mathbf{X}\vec{w}$ for some vector $\vec{w}$ which has norm smaller than one (we'll see later why this is necessary), i.e. the set

$$\widetilde{\mathcal{V}} = \{\widetilde{\vec{v}} = \mathbf{X}\vec{w} : \|\vec{w}\|_2 \leq 1, \vec{w} \in \mathbb{R}^d\} \subset \mathcal{V}.$$

Notice that the gradient of $\mathcal{L}$ with respect to $\vec{v}^t$ need not be in this smaller subspace, i.e. $\nabla_{\vec{v}}\mathcal{L}(\vec{v}^t) \notin \widetilde{\mathcal{V}}$. In that case, we consider a greedy strategy: We first find the direction in $\widetilde{\mathcal{V}}$ which best aligns with the negative gradient. That is at each iteration, we find

$$\widetilde{\vec{v}}^t = \arg\max_{\widetilde{\vec{v}} \in \widetilde{\mathcal{V}}} \langle -\nabla\mathcal{L}(\vec{v}^t), \widetilde{\vec{v}} \rangle. \tag{3}$$

and then compute $\vec{v}^{t+1}$ using the update equation

$$\vec{v}^{t+1} = \vec{v}^t + \alpha_t \widetilde{\vec{v}}^t, \tag{4}$$

for an appropriate choice of step size α_t, where $\widetilde{\vec{v}}^t$ now belongs to the subset $\widetilde{\mathcal{V}}$. We now show how this new procedure yields updates in the same direction as performing the gradient step with respect to $\vec{w}$ explicitly.

Again assume that the current iterate is given by $\vec{v}^t = \mathbf{X}\vec{w}^t$ for some vector $\vec{w}^t$. Now, using equation (3) and the expression for $\nabla_{\vec{v}}\mathcal{L}(\vec{v})\big|_{\vec{v}=\mathbf{X}\vec{w}^t}$ derived previously, **show that $\widetilde{\vec{v}}^t$ in (3) is exactly aligned with the vector $-\mathbf{X}\mathbf{X}^\top(\mathbf{X}\vec{w}^t - \vec{y})$ (i.e. they are the same up to some constant scalar). Now write the update equation (4) in terms of $\mathbf{X}, \vec{w}^t, \vec{y}$ and α_t.** Thus, we have seen that gradient descent on the losses L and $\mathcal{L}$ are essentially the same (and in fact the same if adjusting the stepsize α_t accordingly) by choosing the set $\widetilde{\mathcal{V}}$ accordingly.

Hint: For any function h mapping to a scalar you may use that

$$\arg\max_{\vec{v}=\mathbf{X}\vec{w}:\|\vec{w}\|_2\leq 1} h(\vec{v}) = \mathbf{X}\arg\max_{\|\vec{w}\|_2\leq 1} h(\mathbf{X}\vec{w}).$$

$$\begin{aligned}
&\widetilde{\vec{v}}^t \\
=&\arg\max_{\widetilde{\vec{v}} \in \widetilde{\mathcal{V}}} \langle -\nabla\mathcal{L}(\vec{v}^t), \widetilde{\vec{v}} \rangle \\
=&\mathbf{X}\arg\max_{\|\vec{w}\|_2\leq 1} \langle -\nabla\mathcal{L}(\vec{v}^t), \mathbf{X}\vec{w} \rangle \\
=&\mathbf{X}\arg\max_{\|\vec{w}\|_2\leq 1} \langle \vec{y} - \vec{v}^t, \mathbf{X}\vec{w} \rangle \\
=&\mathbf{X}\arg\max_{\|\vec{w}\|_2\leq 1} \langle \vec{y} - \mathbf{X}\vec{w}^t, \mathbf{X}\vec{w} \rangle
\end{aligned}$$

$$=\mathbf{X}\arg\max_{\|\vec{w}\|_2\leq 1}(\vec{y}-\mathbf{X}\vec{w}^t)^\top\mathbf{X}\vec{w}$$

This is a constrained optimization problem and it looks super convex, so let's consider Lagrangian multiplier. (Note: I cheat here by changing the constraint from $\|\vec{w}\|_2 \leq 1$ to $\|\vec{w}\|_2^2 \leq 1$)

$$L = (\vec{y}-\mathbf{X}\vec{w}^t)^\top\mathbf{X}\vec{w} - \lambda(\|\vec{w}\|_2^2 - 1)$$

$$\nabla_{\vec{w}}L = (\vec{y}-\mathbf{X}\vec{w}^t)^\top\mathbf{X} - 2\lambda\vec{w}$$

Set the derivative to zero gives us:

$$\vec{w} = \frac{1}{2\lambda}(\vec{y}-\mathbf{X}\vec{w}^t)^\top\mathbf{X}$$

This is, when $\vec{w}$ is collinear with $(\vec{y}-\mathbf{X}\vec{w}^t)^\top\mathbf{X}$. Therefore, we have:

$$\begin{aligned}
&\widetilde{\vec{v}}^t\\
=&\mathbf{X}\arg\max_{\|\vec{w}\|_2\leq 1}(\vec{y}-\mathbf{X}\vec{w}^t)^\top\mathbf{X}\vec{w}\\
=&\mathbf{X}\frac{1}{2\lambda}(\vec{y}-\mathbf{X}\vec{w}^t)^\top\mathbf{X}\\
=&\mathbf{X}\frac{1}{2\lambda}\mathbf{X}^\top(\vec{y}-\mathbf{X}\vec{w}^t)\\
=&-\frac{1}{2\lambda}\mathbf{X}\mathbf{X}^\top(\mathbf{X}\vec{w}^t-\vec{y})
\end{aligned}$$

$$\begin{aligned}
\vec{v}^{t+1} &= \vec{v}^t + \alpha_t\widetilde{\vec{v}}^t\\
\vec{v}^{t+1} &= \mathbf{X}\vec{w}^t - \alpha_t\mathbf{X}\mathbf{X}^\top(\mathbf{X}\vec{w}^t-\vec{y})
\end{aligned}$$

(c) Towards Matching Pursuit in the gradient descent framework In this problem we are going to interpret the direction-selection part of Matching Pursuit as a general gradient-type method as described in (b) for an appropriate choice of the set $\widetilde{\mathcal{V}}$.

As you have learned in lecture and discussion, Orthogonal Matching Pursuit is a greedy method to pick relevant coordinates that can explain the data best. It searches for a direction (among the columns) such that the residual error can be explained best and updates the entire fit in the augmented space. Matching Pursuit (MP) is a simpler variant where the entire fit is not updated at every step, just the fit along the new coordinate that was chosen.

Let $\mathbf{X} \in \mathbb{R}^{n \times d}$ be the training feature matrix, $\vec{y}$ the observed training target variables from which we want to infer (potentially sparse) weights $\vec{w} \in \mathbb{R}^d$. At every iteration t, MP picks the column of $\mathbf{X}$ with maximal absolute inner product with the current residual $\vec{r}^t = \vec{y} - \mathbf{X}\vec{w}^t$, i.e.

$$i_t = \arg\max_{i=1,\dots,d} |\langle \vec{r}^t, \vec{\phi}_i(\mathbf{X}) \rangle|. \tag{5}$$

where $\mathbf{X} \in \mathbb{R}^{n \times d}$ is the data matrix with columns $\vec{\phi}_i(\mathbf{X})$ and rows $\vec{x}_1, \dots, \vec{x}_n$, i.e. $\mathbf{X} = (\vec{x}_1, \dots, \vec{x}_n)^\top = (\vec{\phi}_1(\mathbf{X}), \dots, \vec{\phi}_d(\mathbf{X}))$. Note that $\vec{x}_i$ denotes the i-th data point and $\vec{\phi}_i(\mathbf{X})$ denotes the vector of the i-th features of all the data points (i.e. the i-th column of the data matrix $\mathbf{X}$).

The updates of MP at each iteration t then read

$$\mathbf{X}\vec{w}^{t+1} = \mathbf{X}\vec{w}^t + \frac{\langle \vec{r}^t, \vec{\phi}_{i_t}(\mathbf{X}) \rangle}{\|\vec{\phi}_{i_t}(\mathbf{X})\|^2} \vec{\phi}_{i_t}(\mathbf{X}). \tag{6}$$

Now your task in this part is to establish the correspondence between the MP selection rule (5) and the general approach in (restricted) gradient descent given by (3).

In particular, set $\vec{v}^t = \mathbf{X}\vec{w}^t$ and **find the set $\widetilde{\mathcal{V}}$ such that the selection represented by** (5) **can be understood as just being** (3) **in action with $\widetilde{\vec{v}}^t = \mathbf{sign}(\langle \vec{r}^t, \vec{\phi}_{i_t}(\mathbf{X}) \rangle) \vec{\phi}_{i_t}(\mathbf{X})$.**

Hint: don't forget how to deal with the absolute value.

$$\begin{aligned}
& i_t \\
=& \arg\max_{i=1,\dots,d} |\langle \vec{r}^t, \vec{\phi}_i(\mathbf{X}) \rangle| \\
=& \arg\max_{i=1,\dots,d} \operatorname{sign}(\langle \vec{r}^t, \vec{\phi}_i(\mathbf{X}) \rangle) \langle \vec{r}^t, \vec{\phi}_i(\mathbf{X}) \rangle \\
=& \arg\max_{i=1,\dots,d} \operatorname{sign}(\langle \vec{r}^t, \vec{\phi}_i(\mathbf{X}) \rangle) \langle \vec{y} - \mathbf{X}\vec{w}^t, \vec{\phi}_i(\mathbf{X}) \rangle \\
=& \arg\max_{i=1,\dots,d} \operatorname{sign}(\langle \vec{r}^t, \vec{\phi}_i(\mathbf{X}) \rangle) (\vec{y} - \mathbf{X}\vec{w}^t)^\top \vec{\phi}_i(\mathbf{X}) \\
=& \arg\max_{i=1,\dots,d} (\vec{y} - \mathbf{X}\vec{w}^t)^\top \operatorname{sign}(\langle \vec{r}^t, \vec{\phi}_i(\mathbf{X}) \rangle) \vec{\phi}_i(\mathbf{X}) \\
=& \arg\max_{i=1,\dots,d} \langle \vec{y} - \mathbf{X}\vec{w}^t, \operatorname{sign}(\langle \vec{r}^t, \vec{\phi}_i(\mathbf{X}) \rangle) \vec{\phi}_i(\mathbf{X}) \rangle
\end{aligned}$$

Copy equation (3) from above and the fact that we want $\widetilde{\vec{v}} = \operatorname{sign}(\langle \vec{r}^t, \vec{\phi}_{i_t}(\mathbf{X}) \rangle) \vec{\phi}_{i_t}(\mathbf{X})$. This tells us we need to constraint $\widetilde{\vec{v}}$ to be collinear with one of the column of $\mathbf{X}$ as $\vec{\phi}_{i_i}(\mathbf{X})_t$. Therefore, we can rewrite $\widetilde{\mathcal{V}}$ to be:

$$\widetilde{\mathcal{V}} = \{\widetilde{\vec{v}} = \mathbf{X}\vec{w} : \|\vec{w}\|_0 = 1 \text{ and } \sum_{m=1}^{d} w_m = \operatorname{sign}(\langle \vec{r}^t, \vec{\phi}_{i_t}(\mathbf{X}) \rangle), \vec{w} \in \mathbb{R}^d\} \subset \mathcal{V}$$

we can rewrite this to be:

$$\widetilde{\mathcal{V}} = \{\widetilde{\vec{v}} = \mathrm{sign}(\langle \vec{r}^t, \vec{\phi}_i(\mathbf{X})\rangle)\vec{\phi}_i(\mathbf{X}) : i = 1, \ldots, d\} \subset \mathcal{V}$$

$$\widetilde{\vec{v}}^t = \arg\max_{\widetilde{\vec{v}} \in \widetilde{\mathcal{V}}} \langle -\nabla \mathcal{L}(\vec{v}^t), \widetilde{\vec{v}} \rangle$$

here, we substitute in $\widetilde{\mathcal{V}}$

$$\widetilde{\vec{v}}^t = \arg\max_{\vec{\phi}_i(\mathbf{X}) \in \{\vec{\phi}_1(\mathbf{X}), \vec{\phi}_2(\mathbf{X}), \ldots, \vec{\phi}_d(\mathbf{X})\}} \langle (\vec{y} - \mathbf{X}\vec{w}^t), \mathrm{sign}(\langle \vec{r}^t, \vec{\phi}_i(\mathbf{X})\rangle)\vec{\phi}_i(\mathbf{X})\rangle$$

We can see this is already the same form as the selection represented by (5). The only difference is that here we returns a vector not its index. Therefore, we have found a new set $\widetilde{\mathcal{V}}$ that (5) is equivalent to (3). Intuitively, it means we search for the best column and do gradient descent on that direction which can decrease our residue.

(d) In traditional gradient descent, the rule for choosing the step-size α_t is left open. One way of doing this is adaptively by *linesearch* — picking the step-size α that reduces the loss the most. Formally, choosing stepsize α_t via linesearch with respect to the search direction $\widetilde{\vec{v}}^t$ at each iteration for a loss $\mathcal{L}$ means finding the best α such that $\alpha_t = \arg\min_{\alpha \geq 0} \mathcal{L}(\vec{f}^t(\{\vec{x}\}_{i=1}^n) + \alpha \widetilde{\vec{v}}^t)$. **Argue that the update (4) with $\widetilde{\vec{v}}^t$ computed using equation (3) and linesearch for α_t is the same as doing the MP update (6).**

Copy the solution to part (b) from above:

$$\begin{aligned}
\vec{v}^{t+1} &= \vec{v}^t + \alpha_t \widetilde{\vec{v}}^t \\
\vec{v}^{t+1} &= \mathbf{X}\vec{w}^t - \alpha_t \mathbf{X}\mathbf{X}^\top(\mathbf{X}\vec{w}^t - \vec{y})
\end{aligned}$$

Apply the new set $\widetilde{\mathcal{V}}$ we found it becomes:

$$\begin{aligned}
\vec{v}^{t+1} &= \vec{v}^t + \alpha_t \widetilde{\vec{v}}^t \\
\vec{v}^{t+1} &= \mathbf{X}\vec{w}^t - \alpha_t \text{sign}(\langle \vec{r}^t, \vec{\phi}_{i_t}(\mathbf{X})\rangle)\vec{\phi}_i(\mathbf{X})
\end{aligned}$$

Copy the MP update (6) from above:

$$\begin{aligned}
\mathbf{X}\vec{w}^{t+1} &= \mathbf{X}\vec{w}^t + \frac{\langle \vec{r}^t, \vec{\phi}_{i_t}(\mathbf{X})\rangle}{\|\vec{\phi}_{i_t}(\mathbf{X})\|^2}\vec{\phi}_{i_t}(\mathbf{X}) \\
\mathbf{X}\vec{w}^{t+1} &= \mathbf{X}\vec{w}^t + \frac{\langle \vec{y} - \mathbf{X}\vec{w}^t, \vec{\phi}_{i_t}(\mathbf{X})\rangle}{\|\vec{\phi}_{i_t}(\mathbf{X})\|^2}\vec{\phi}_{i_t}(\mathbf{X}) \\
\mathbf{X}\vec{w}^{t+1} &= \mathbf{X}\vec{w}^t + \frac{\vec{\phi}_{i_t}(\mathbf{X})^\top(\vec{y} - \mathbf{X}\vec{w}^t)}{\|\vec{\phi}_{i_t}(\mathbf{X})\|^2}\vec{\phi}_{i_t}(\mathbf{X})
\end{aligned}$$

We notice that if we can find an α to make two equations the same:

$$\alpha_t = -\text{sign}(\langle \vec{r}^t, \vec{\phi}_{i_t}(\mathbf{X})\rangle)\frac{\vec{\phi}_{i_t}(\mathbf{X})^\top(\vec{y} - \mathbf{X}\vec{w}^t)}{\|\vec{\phi}_{i_t}(\mathbf{X})\|^2}$$

We can get this solution by solving the optimization problem given.
Here our lost function is simply $l2$-norm loss, which is represented by the residue error. Because the question only asks for an argument instead of a proof, I will write it in English.
In the MP update, we have already found the direction that can minimize our loss, which is $\vec{\phi}_{i_t}(\mathbf{X})$. If we have found that direction in the update of $\widetilde{\vec{v}}^t$, then the only thing we are missing is the α_t. However, if there is no constraint, the direction found in the update of $\widetilde{\vec{v}}^t$ might be linear combinations of columns instead of one single column $\vec{\phi}_{i_t}(\mathbf{X})$. By setting the constrain and limit the search to the new set $\widetilde{\mathcal{V}}$, we can change it to a linear search after determining the direction.

(e) (BONUS) *MP as gradient boosting* The two previous greedy principles of matching-pursuit (column selection and linesearch) can be extended to more general settings of non-linear and non-parametric function classes $\mathcal{F}$ and general loss functions $\mathcal{L}$ (which need not be squared loss). Consider the general learning problem where again we are given a bunch of pairwise training samples in the form of $(\vec{x}_i, y_i)$ for $i = 1, \ldots, n$, with $\vec{x}_i \in \mathbb{R}^d$ and $y_i \in \mathcal{Y} \subset \mathbb{R}$ and we want to learn a function $f : \mathbb{R}^d \to \mathcal{Y}$ from a function space $\mathcal{F}$ (with possibly non-linear and non-parametric basis elements, such as trees) which we can then use to predict y_{test} by $f(\vec{x}_{test})$ for a test sample $\vec{x}_{test}$.

For this purpose we minimize the following (average) loss $\mathcal{L} : \mathbb{R}^n \to \mathbb{R}$

$$\mathcal{L}(\vec{f}(\{\vec{x}\}_{i=1}^n)) = \mathcal{L}(f(\vec{x}_1), \ldots, f(\vec{x}_n)) = \frac{1}{n}\sum_{i=1}^n l(y_i, f(\vec{x}_i)).$$

for some point-wise loss function $l : \mathcal{Y} \times \mathcal{Y} \mapsto \mathbb{R}$ for one sample. Notice that the loss maps a vector to a scalar, as in the linear settings. Here, we use $\vec{f}(\{\vec{x}\}_{i=1}^n) := (f(\vec{x}_1), \ldots, f(\vec{x}_n))$ to denote the vector of function values of $f \in \mathcal{F}$ at the points $\vec{x}_1, \ldots, \vec{x}_n$.

As in gradient descent (4), we want to minimize the loss using an iterative algorithm with updates of the form

$$f^{t+1} = f^t + \alpha_t \widetilde{f}^t.$$

Note that our notation follows the convention that $f^t \in \mathcal{F}$ is the current iterate and $\widetilde{f}^t \in \mathcal{W}$ is the direction that we are adding at each step. The main question is how to choose $\widetilde{f}^t$. Let's assume that it is for example computationally much simpler to search over a smaller set of functions $\mathcal{W} \subset \mathcal{F}$ at each timestep.

Similar to the parametric update in (3), we then choose the function $\widetilde{f} \in \mathcal{W}$ for which the function value vector $\vec{\widetilde{f}}(\{\vec{x}\}_{i=1}^n)$ has the maximal inner product with the negative gradient, i.e.

$$\widetilde{f}^t = \arg\max_{\widetilde{f} \in \mathcal{W}} \langle -\nabla\mathcal{L}(\vec{f}^t(\{\vec{x}\}_{i=1}^n)), \vec{\widetilde{f}}(\{\vec{x}\}_{i=1}^n)\rangle \tag{7}$$

where the gradient $\nabla\mathcal{L}(\vec{f}^t(\{\vec{x}\}_{i=1}^n)), \vec{\widetilde{f}}(\{\vec{x}\}_{i=1}^n) = \nabla_{\vec{v}}\mathcal{L}(\vec{v})\big|_{\vec{v}=\vec{f}^t(\{\vec{x}\}_{i=1}^n)}$ which you have already computed above for the square loss.

Now we want to see how Matching Pursuit can be viewed as an instance of this framework, also called *gradient boosting* or *functional gradient descent*. Specifically, **show that the particular choices of**

$$\mathcal{F} = \{f(\vec{x}) = \vec{w}^\top \vec{x} : \vec{w} \in \mathbb{R}^d\} \ \textbf{and} \ \ \mathcal{W} = \{\widetilde{f}(\vec{x}) = \pm\vec{e}_i^\top \vec{x} : i = 1, \ldots, d\},$$

where $\vec{e}_i$ is the i-th standard basis element with a 1 in the i-th position and zeros elsewhere, yield Matching Pursuit as a form of gradient boosting. Notice that $\mathcal{F}$ is the function space of all linear functions and $\mathcal{W}$ is the set of functions on vectors $\vec{x} \in \mathbb{R}^d$ which pick out signed coordinates of $\vec{x}$.

Hint: For matching pursuit, what are $\vec{\widetilde{f}}^t(\{\vec{x}\}_{i=1}^n)$ and $\vec{f}^t(\{\vec{x}\}_{i=1}^n)$?

(f) (BONUS) *AdaBoost as gradient boosting* Now let's have a look at non-linear and non-parametric functions such as decision trees for classification. In this problem we will establish that AdaBoost is also an instance of gradient boosting on a particular loss function (which is not the zero-one loss!).

Notice how each decision tree can be viewed as a function $\widetilde{f} : \mathbb{R}^d \to \{-1, +1\}$. Since in each iteration we fit a tree to the weighted samples, our function set $\mathcal{W}$ now corresponds to the set of decision trees with a fixed set of parameters.

Remember the AdaBoost algorithm from lecture and the last homework. At each iteration t, it upweights each sample i by a different w_i^t in every iteration and finds a new tree which solves the reweighted classification problem

$$\widetilde{f}^t_{Ada} = \arg\min_{\widetilde{f}\in\mathcal{W}} \sum_{i=1}^{n} w_i^t \mathbb{I}(y_i \neq \widetilde{f}(\vec{x}_i)) \text{ where } w_i^t = \frac{\exp(-y_i f^t(\vec{x}_i))}{\sum_{i=1}^{n} \exp(-y_i f^t(\vec{x}_i))}$$

where f^t is a linear combination of functions in $\mathcal{W}$ found by previous iterations and $\mathbb{I}(A) = 1$ if A is true and 0 otherwise (the indicator function). Notice that the samples which the current classifier f^t gets wrong are assigned a larger weight.

The update for the overall classifier reads

$$f^{t+1} = f^t + \alpha_t \widetilde{f}^t_{Ada} \tag{8}$$

The classification rule is then obtained by taking the sign of the final iterate at time τ, i.e. $f_{\text{final}}(\vec{x}) = \text{sign}(f^\tau(\vec{x}))$ which lies in a bigger space $\mathcal{F} \supset \mathcal{W}$.

Show that $\vec{\widetilde{f}}^t_{Ada}(\{\vec{x}\}_{i=1}^n) = \vec{\widetilde{f}}^t(\{\vec{x}\}_{i=1}^n)$ **in the gradient boosting framework** (7) **for the loss** $l(y, f(\vec{x})) = \mathbf{e}^{-yf(\vec{x})}$. This implies that finding the best fit for the reweighted samples is equivalent to fitting the gradient. For the gradient in (7), notice that we again view $\mathcal{L}$ as a function of a vector $\vec{v}$ and the gradient $\nabla\mathcal{L}(\vec{f}^t(\{\vec{x}\}_{i=1}^n)) = \nabla_{\vec{v}}\mathcal{L}(\vec{v})\big|_{\vec{v}=\vec{f}^t(\{\vec{x}\}_{i=1}^n)}$.

Hint: Now note that for $y \in \{-1, +1\}$ and $\widetilde{f} : \mathbb{R}^d \to \{-1, +1\}$, we have

$$y_i \widetilde{f}(\vec{x}_i) = 2(1 - \mathbb{I}(y_i \neq \widetilde{f}(\vec{x}_i))) - 1. \tag{9}$$

Hint: Recall that the maximizer does not change if you scale by or add to the function a constant factor which is independent of the object you are searching over, i.e.

$$\arg\max_{\widetilde{f}\in\mathcal{W}} CH(\vec{\widetilde{f}}(\{\vec{x}\}_{i=1}^n)) + G(\vec{f}(\{\vec{x}\}_{i=1}^n)) = \arg\max_{\widetilde{f}\in\mathcal{W}} H(\vec{\widetilde{f}}(\{\vec{x}\}_{i=1}^n))$$

for some functions H, G that depend on the function value vectors $\vec{f}, \vec{\widetilde{f}}$, and a scalar $C \in \mathbb{R}$. $\vec{f}$ does not depend on $\widetilde{f}$.

(g) (BONUS) *AdaBoost weight as linesearch stepsize* In this problem we show that the weights used in the AdaBoost algorithm at each time step t correspond to the "best" stepsize in the direction $\widetilde{f}^t_{Ada}$ using linesearch.

Let's define the error of the classifier $\widetilde{f}^t_{Ada}$ as $\epsilon_t = \sum_{i=1}^n w_i^t \mathbb{I}(y \neq \widetilde{f}^t_{Ada}(\vec{x}_i))$. In practice, AdaBoost uses the stepsize $\alpha_t = \frac{1}{2} \ln \frac{1-\epsilon_t}{\epsilon_t}$.

Show that the AdaBoost stepsize at any time t corresponds to the stepsize chosen via linesearch for the direction $\widetilde{\vec{v}} = \vec{\widetilde{f}}^t_{Ada}(\{\vec{x}\}_{i=1}^n)$.

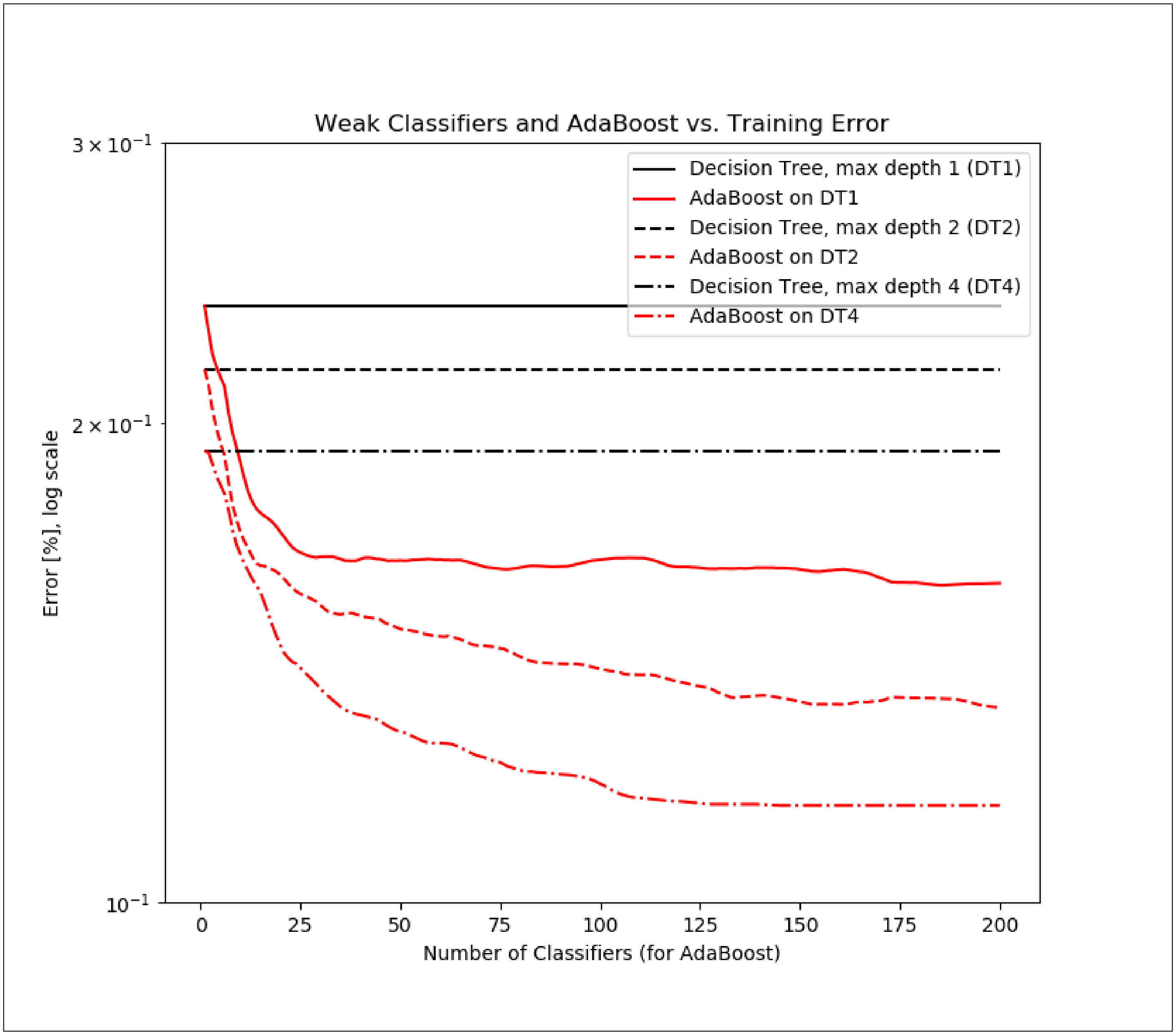
(h) Now we will explore how gradient boosting does in practice for both examples we have theoretically studied in the previous sections: Matching Pursuit and AdaBoost. In particular, we will see how the number of updates, equivalent to the number of trees that we are fitting, effects the test and training error.

We have written some code (see the file: `adaboost.py`) that trains several classifiers on the Spam dataset you worked with last week. **Run the code for this part of the question.** It generates a training error plot that uses several different decision trees (depths 1, 2, and 4) as well as several AdaBoost-trained classifiers, each built off of one of these decision tree classifiers. **Do you see a trend in the performance of different trees by themselves? Do you observe a trend in the training error as you use deeper trees for AdaBoost? Why might this happen?**

Do you see a trend in the performance of different trees by themselves? Yes, I do. AdaBoost generally performs better than Decision Tree alone.

Do you observe a trend in the training error as you use deeper trees for AdaBoost? Yes, I do. AdaBoost performs better when the number of classifier increases.

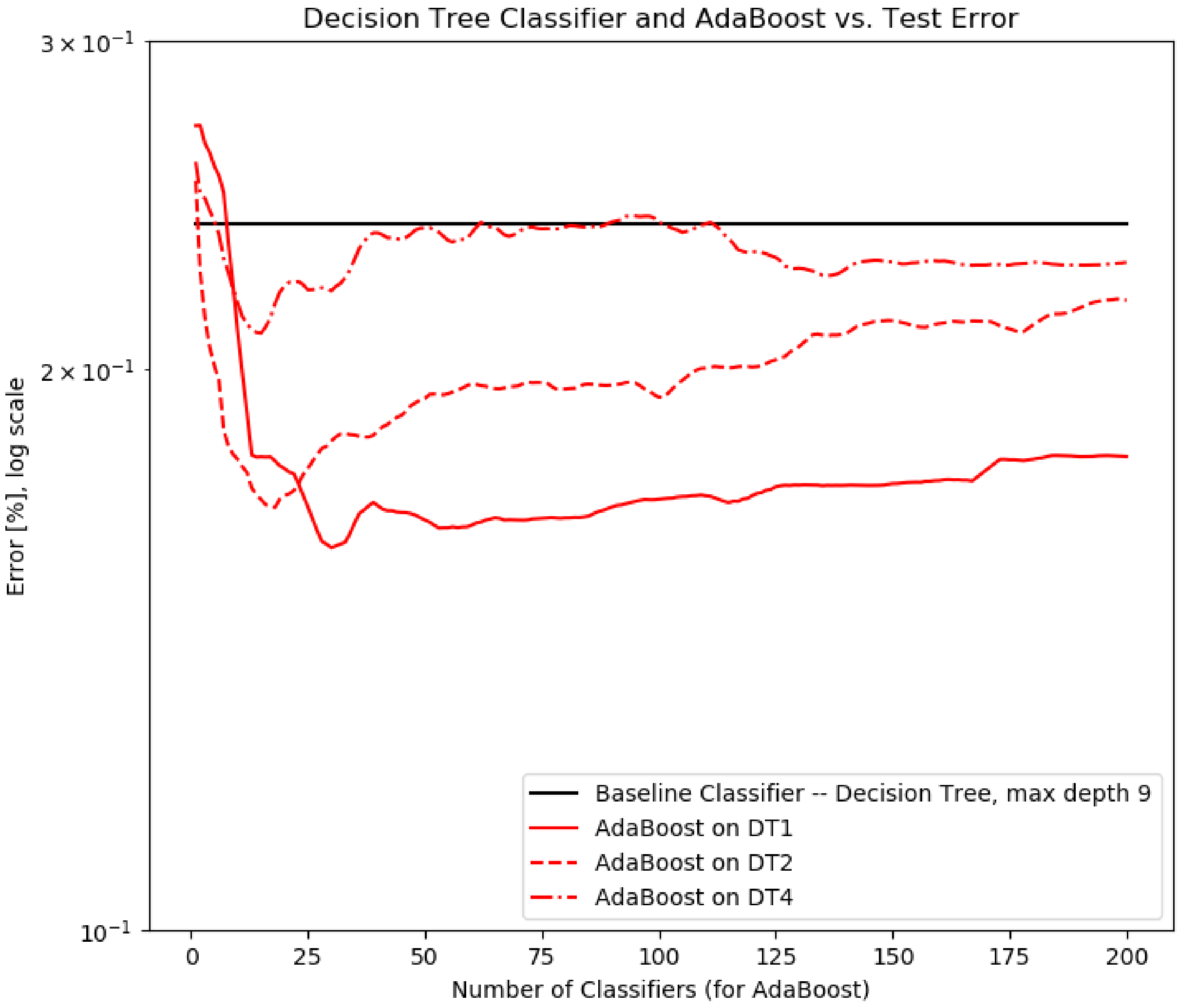
Why might this happen? A bunch of idiots is better than one expert. AdaBoost handles many difficult data points that a single tree cannot handle.


Weak Classifiers and AdaBoost vs. Training Error
Decision Tree, max depth 1 (DT1)
AdaBoost on DT1
Decision Tree, max depth 2 (DT2)
AdaBoost on DT2
Decision Tree, max depth 4 (DT4)
AdaBoost on DT4
3×10^{-1}
2×10^{-1}
10^{-1}
Error [%], log scale
0
25
50
75
100
125
150
175
200
Number of Classifiers (for AdaBoost)

(i) Now we examine the test error, and compare against a baseline classifier (a decision tree of depth 9). **Run the code for this part. Is there a difference in the training and test error? Which decision tree depth works best for AdaBoost? Explain your observations.**

Is there a difference in the training and test error? Yes. In the training data, the error of AdaBoost always decreases as the number of trees increase. This is because more trees can be use to handle the difficult data points. However, in test dataset, we have bias-variance tradeoff. More trees can overfit the training data because even the most difficult data can be tackled if we have infinite number of trees. That's the reason we see a best number of classifiers and after that the test errors start to increase for AdaBoost.

Which decision tree depth works best for AdaBoost? Explain your observations. AdaBoost on DT1 works the best. This is also due to bias-variance tradeoff. The deeper the trees, the lower the bias. However, it also increases the variance. Simple tree works better in this case, because it has small variance with acceptable bias and small overall error.


Decision Tree Classifier and AdaBoost vs. Test Error
3×10^{-1}
2×10^{-1}
10^{-1}
Error [%], log scale
0
25
50
75
100
125
150
175
200
Number of Classifiers (for AdaBoost)
Baseline Classifier -- Decision Tree, max depth 9
AdaBoost on DT1
AdaBoost on DT2
AdaBoost on DT4

(j) In the last part, you noticed that the test error decreases as a function of boosting iterations in the beginning but eventually it starts to increase when the number of decision trees in Adaboost is pretty large. In practice, this phenomenon motivates us to stop training early and limit the number of classifiers used in a boosting setting. Here “stopping early” means that training error has not reduced to zero but we have stopped training! Refer to the plot from the previous part and answer: **Do you think limiting the number of base classifiers used for AdaBoost would help? Which base classifier can we run more boosting iterations on before the test error starts increasing? Justify your answer intuitively.**

Do you think limiting the number of base classifiers used for AdaBoost would help?

Yes, it wound help. Stopping early can avoid overfitting on the training data.

Which base classifier can we run more boosting iterations on before the test error starts increasing? Justify your answer intuitively. We can run more boosting iterations on the classifier with max depth 1 before the test error starts increasing. This can be seen on the graph in the previous part. Intuitively, decision trees with max depth 1 has very large bias when the number of trees is small because it cannot classify data points well with limited number of splits. To compensate for that, we need more trees to make it smarter.

(k) In this part, we connect this phenomenon to matching pursuit. The provided code generates a synthetic dataset:

$$\vec{y}_{\text{train}}, \vec{y}_{\text{test}} \in \mathbb{R}^n$$

$$\mathbf{X}_{\text{train}}, \mathbf{X}_{\text{test}} \in \mathbb{R}^{n \times d}$$

$$\vec{w} \in \mathbb{R}^d,$$

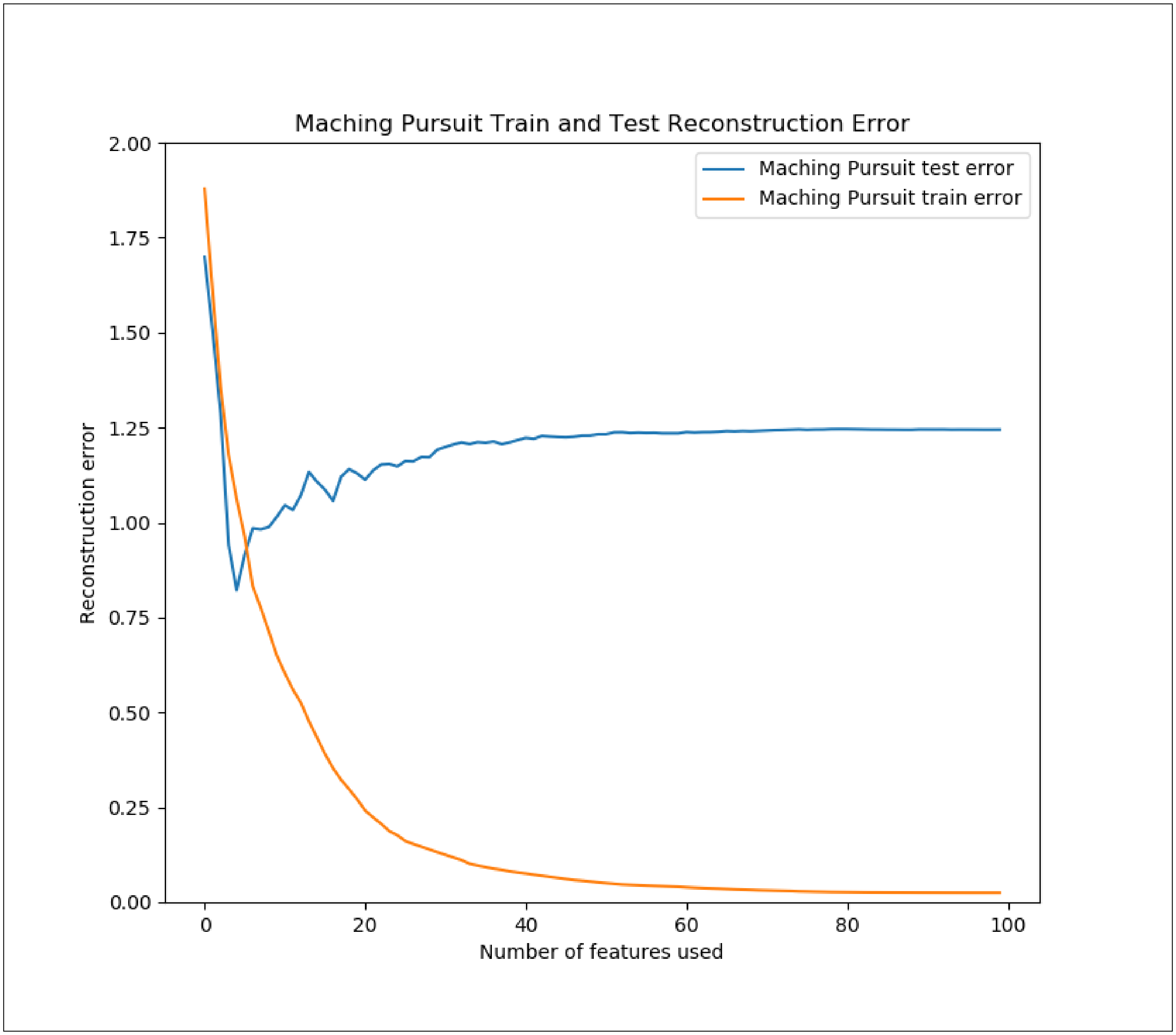
with (for both test and train data)

$$\vec{y} = \mathbf{X}\vec{w} + \vec{z}.$$

Here, $\vec{w}$ is a sparse vector (it has only a few non-zero entries) and $\vec{z}$ denotes noise. Given the observations $\vec{y}_{\text{train}}$ and the feature matrix $\mathbf{X}_{\text{train}}$, we are tasked with finding a sparse solution $\hat{\vec{w}}$ for which we use the matching pursuit algorithm.

Let $\hat{\vec{w}}^t_{\text{MP}}$ denote the estimate of $\vec{w}$ recovered by MP after t-iterations. The code also plots the training error $\|\vec{y}_{\text{train}} - \mathbf{X}_{\text{train}}\hat{\vec{w}}^t_{\text{MP}}\|^2$ and the test error $\|\vec{y}_{\text{test}} - \mathbf{X}_{\text{test}}\hat{\vec{w}}^t_{\text{MP}}\|^2$ as a function of iterations t (as t increases matching pursuit builds an estimate using a greater number of features). **Run the code for this part. Explain the shape of the training error plot. Does the plot for test error look similar to the one from part (h)? Comment on the similarities.** You may use conclusions obtained in previous parts to justify your comments.

Yes. the shape of the plot looks similar to the one from part (h). They are similar because $\vec{w}$ is a sparse vector and adding more features (bad features) will increase the variance which increases the overall error.


Maching Pursuit Train and Test Reconstruction Error
Maching Pursuit test error
Maching Pursuit train error
Reconstruction error
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75
2.00
0
20
40
60
80
100
Number of features used

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.io import loadmat

from sklearn.datasets import make_sparse_coded_signal
from sklearn.ensemble import AdaBoostClassifier, RandomForestClassifier
from sklearn.metrics import zero_one_loss
from sklearn.tree import DecisionTreeClassifier

# globals
n_estimators = 200
DT1 = DecisionTreeClassifier(max_depth=1, min_samples_leaf=15)
DT2 = DecisionTreeClassifier(max_depth=2, min_samples_leaf=15)
DT4 = DecisionTreeClassifier(max_depth=4, min_samples_leaf=15)
DT9 = DecisionTreeClassifier(max_depth=9, min_samples_leaf=1)

"""Loads the training data from the SPAM dataset used in HW12."""
def load_data():
# load data
data = loadmat("datasets/spam_data/spam_data.mat")
# training data
data_, labels_ = data["training_data"], np.squeeze(data["training_labels"])
X_train, y_train = data_, labels_
# test data
y_test=[]
with open("datasets/spam_data/spam_test_labels.txt","r") as f:
for l in f.readlines():
y_test.append(int(l.split(",")[1]))
y_test = np.array(y_test)
X_test = data[’test_data’]

return X_train, y_train, X_test, y_test

"""Runs the maching pursuit algorithm."""
def mp(y, X, w_true, y_test, X_test):
train_err = []
test_err = []
X_ = X
y = np.copy(y); X = np.copy(X)
curr = np.copy(y)
w_est = np.zeros(len(X[0]))
for j in range(len(X[0])):
i = np.argmax(np.abs(np.dot(X.T, curr)))
col = np.copy(X[:,i])
# use each column only once
X[:,i] = 0
w_est[i] = np.dot(col, curr)
curr = curr - col*w_est[i]
# error defined here as ||y - D x_hat||_2
train_err.append(np.linalg.norm(X_.dot(w_est) - y))
test_err.append(np.linalg.norm(X_test.dot(w_est)-y_test))

return w_est, train_err, test_err

if __name__ == "__main__":
###### CHANGE THESE VARIABLES TO RUN PROBLEM PARTS
PART_G = True
PART_H = True
PART_J = True
######
X_train, y_train, X_test, y_test = load_data()

### PART G
```

```
if PART_G:
fig = plt.figure(figsize=(8,7))
ax = fig.add_subplot(111)
styles=["k-", "k--", "k-."]
depths=[1,2,4]
j=0
# for each weak classifier, train it and an AdaBoost instance based on it
for w in [DT1, DT2,DT4]:
# Weak classifier
w.fit(X_train, y_train)
err = 1.0 - w.score(X_train, y_train)
ax.plot([1, n_estimators], [err] * 2, styles[j],
label="Decision Tree, max depth %d (DT%d)" % (depths[j],depths[j]))
# AdaBoost classifier
ada = AdaBoostClassifier(base_estimator=w,
n_estimators=n_estimators,
random_state=0)

ada_train_err = np.zeros((n_estimators,))
ada.fit(X_train, y_train)
for i, y_pred in enumerate(ada.staged_predict(X_train)):
ada_train_err[i] = zero_one_loss(y_pred, y_train)

smoothed = []
# use moving average filter to smooth plots -- done to make easier
# to see trends; you are encouraged to also plot ’ada_train_err’ to
# see the actual error plots!!
for i in range(len(ada_train_err)):
temp = 0.
counter = 0.
for k in range(i-5, i+1):
if k >= 0:
temp += ada_train_err[k]
counter += 1.
smoothed.append(temp/counter)

ax.plot(np.arange(n_estimators) + 1, smoothed, styles[j],
label="AdaBoost on DT%d" % depths[j],
color="red")

j += 1

ax.set_ylim((0.1, 0.3))
ax.set_yscale(’log’)
ax.set_xlabel("Number of Classifiers (for AdaBoost)")
ax.set_ylabel("Error [%], log scale")
ax.set_title("Weak Classifiers and AdaBoost vs. Training Error")
leg = ax.legend(loc=’upper right’, fancybox=True)
leg.get_frame().set_alpha(0.7)
#plt.show()
plt.savefig(’Figure_2g’)
plt.close()

### PART H
if PART_H:
fig = plt.figure(figsize=(8,7))
ax = fig.add_subplot(111)
# Basline classifier (a "deep" tree)
DT9.fit(X_train, y_train)
err = 1.0 - DT9.score(X_test, y_test)
ax.plot([1, n_estimators], [err] * 2, "k-",
label="Baseline Classifier -- Decision Tree, max depth 9")
# AdaBoost
```

```
styles=["k-", "k--", "k-."]
depths=[1,2,4]
j=0
# for each weak classifier, train an AdaBoost instance based on it
for w in [DT1, DT2, DT4]:

# AdaBoost classifier
ada = AdaBoostClassifier(base_estimator=w,
n_estimators=n_estimators,
random_state=0)
ada_train_err = np.zeros((n_estimators,))

ada.fit(X_train, y_train)
for i, y_pred in enumerate(ada.staged_predict(X_test)):
ada_train_err[i] = zero_one_loss(y_pred, y_test)

smoothed = []
# use moving average filter to smooth plots -- done to make easier
# to see trends; you are encouraged to also plot 'ada_train_err' to
# see the actual error plots!!
for i in range(len(ada_train_err)):
temp = 0.
counter = 0.
for k in range(i-5, i+1):
if k >= 0:
temp += ada_train_err[k]
counter += 1.
smoothed.append(temp/counter)

ax.plot(np.arange(n_estimators) + 1, smoothed, styles[j],
label="AdaBoost on DT%d" % depths[j],
color="red")

j += 1

ax.set_ylim((0.1, 0.3))
ax.set_yscale('log')
ax.set_xlabel("Number of Classifiers (for AdaBoost)")
ax.set_ylabel("Error [%], log scale")
ax.set_title("Decision Tree Classifier and AdaBoost vs. Test Error")
leg = ax.legend(loc='lower right', fancybox=True)
leg.get_frame().set_alpha(0.7)
#plt.show()
plt.savefig('Figure_2h')
plt.close()

### PART J
if PART_J:
fig = plt.figure(figsize=(8,7))
ax = fig.add_subplot(111)

n_components = 100
n_features = 30
n_nonzero_coefs = 5
# y = Xw; w is a sparse vector
y_train, X_train, w = make_sparse_coded_signal(n_samples=1,
n_components=n_components,
n_features=n_features,
n_nonzero_coefs=n_nonzero_coefs,
random_state=0)
# test set
_, X_test, _ = make_sparse_coded_signal(n_samples=1,
n_components=n_components,
n_features=n_features,
n_nonzero_coefs=n_nonzero_coefs,
random_state=0)
y_test = np.dot(X_test, w)

np.random.seed(10)
y_noised_train = y_train + 2e-1*np.random.randn(len(y_train))
y_noised_test = y_test + 2e-1*np.random.randn(len(y_test))
w_est, train_err, test_err = mp(y_noised_train, X_train, w,
y_noised_test, X_test)

ax.plot(np.arange(n_components), test_err, label="Maching Pursuit test error")
ax.plot(np.arange(n_components), train_err, label="Maching Pursuit train error")
ax.set_ylim((0., 2.0))
ax.set_xlabel("Number of features used")
ax.set_ylabel("Reconstruction error")
ax.set_title("Maching Pursuit Train and Test Reconstruction Error")

leg = ax.legend(loc='upper right', fancybox=True)
leg.get_frame().set_alpha(0.7)

#plt.show()
plt.savefig('Figure_2j')
plt.close()
```

Question 3. CNNs on Fruits and Veggies

In this problem, we will use the dataset of fruits and vegetables that was collected in HW5. The goal is to accurately classify the produce in the image. In prior homework, we explored how to select features and then use linear classification to learn a function. We will now explore using Convolutional Neural Networks to optimize feature selection jointly with learning a classification policy.

Denote the input state $x \in \mathbb{R}^{90\times 90\times 3}$, which is a down sampled RGB image with the fruit centered in it. Each data point will have a corresponding class label, which corresponds to their matching produce. Given 25 classes, we can denote the label as $y \in \{0, ..., 24\}$.

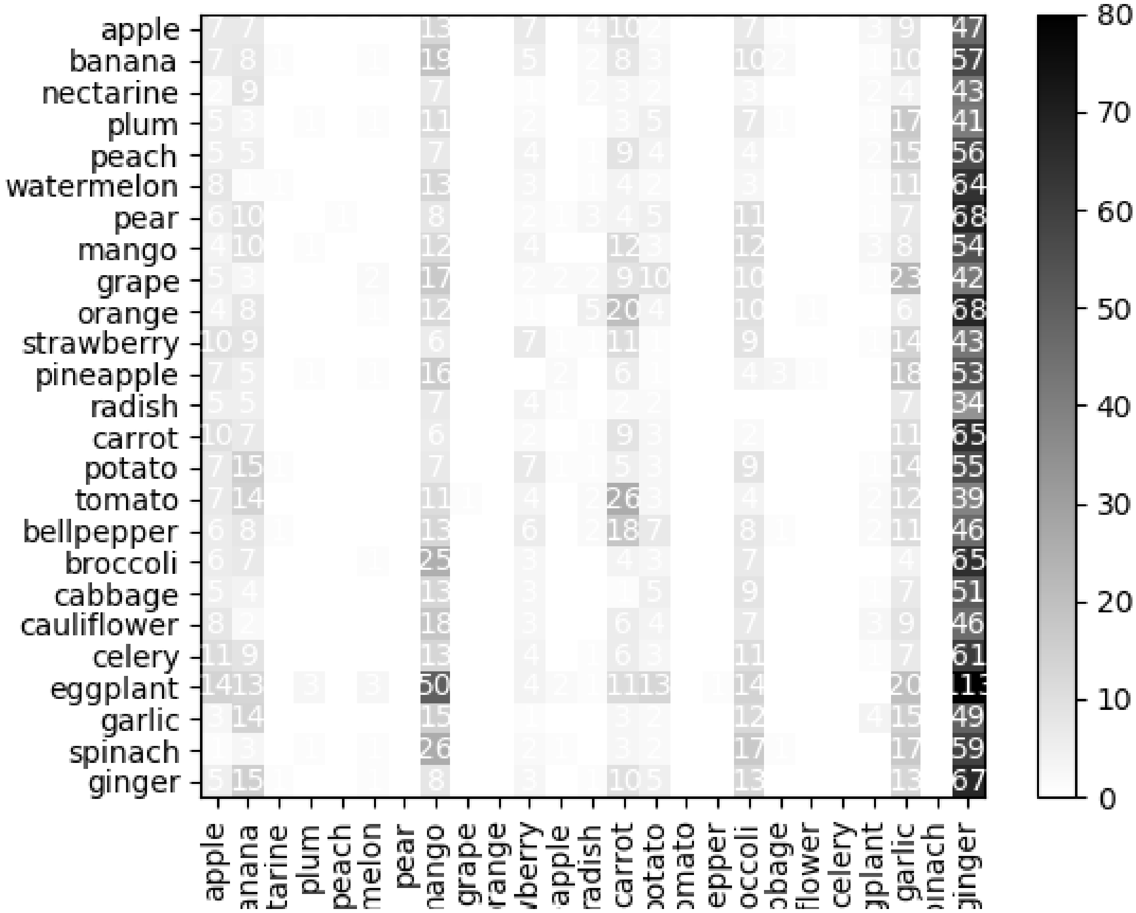
The goal of this problem is twofold. First you will learn how to implement a Convolutional Neural Network (CNN) using TensorFlow. Then we will explore some of the mysteries about why neural networks work as well as what they do in the context of a bias variance trade-off.

Note all python packages needed for the project, will be imported already. **DO NOT import new Python libraries.** Also, this project will be computationally expensive on your computer's CPU. Please use the free EC2 credit if you do not have a strong computer. The instructions for EC2 are on Piazza thread @391. The dataset for this project can be download here(link). We recommend using Tensorflow version 1.6.0 with Python3 interface.

(a) To begin the problem, we need to implement a CNN in TensorFlow. In order to reduce the burden of implementation, we are going to use a TensorFlow wrapper known as *slim*. In the starter code is a file named *cnn.py*, the network architecture and the loss function are currently blank. Using the slim library, you will have to write a convolutional neural network that has the following architecture:

1. Layer 1: A convolutional layer with 5 filters of size 15 by 15
2. Non-Linear Response: Rectified Linear Units
3. A max pooling operation with filter size of 3 by 3
4. Layer 2: A Fully Connected Layer with output size 512.
5. Non-Linear Response: Rectified Linear Units
6. Layer 3: A Fully Connected Layer with output size 25 (i.e. the class labels)
7. Loss Layer: Softmax Cross Entropy Loss

In the file *example_cnn.py*, we show how to implement a network in TensorFlow Slim. Please use this as a reference. Once the network is implemented **run the script** *test_cnn_part_a.py* **on the dataset and report the resulting confusion matrix**. The goal is to ensure that your network compiles, but we should not expect the results to be good because it is randomly initialized.



```
import tensorflow as tf

# import yolo.config_card as cfg

slim = tf.contrib.slim


class CNN(object):
def __init__(self, classes, image_size):
'''
Initializes the size of the network
'''

self.classes = classes
self.num_class = len(self.classes)
self.image_size = image_size

self.output_size = self.num_class
self.batch_size = 40

self.images = tf.placeholder(tf.float32, [None, self.image_size, self.image_size, 3], name='images')

self.logits = self.build_network(self.images, num_outputs=self.output_size)

self.labels = tf.placeholder(tf.float32, [None, self.num_class])

self.loss_layer(self.logits, self.labels)
self.total_loss = tf.losses.get_total_loss()
tf.summary.scalar(’total_loss’, self.total_loss)


def build_network(self,
images,
num_outputs,
scope=’yolo’):
with tf.variable_scope(scope):
with slim.arg_scope([slim.conv2d, slim.fully_connected],
weights_initializer=tf.truncated_normal_initializer(0.0, 0.01),
weights_regularizer=slim.l2_regularizer(0.0005)):
’’’
Fill in network architecutre here
Network should start out with the images function
Then it should return net
’’’

###SLIM BY DEFAULT ADDS A RELU AT THE END OF conv2d and fully_connected

###SLIM SPECIFYING A CONV LAYER WITH 5 FILters as SIZE 15 by 15
net = slim.conv2d(images, 5, [15, 15], scope=’conv_0’)

###SAVE RESPONSE MAP HERE FOR VIZ_FEATURES
self.response_map = net

### SLIM USING POOLING ON THE NETWORK. THE POOLING REGION CONSIDERED IS 3 by 3
net = slim.max_pool2d(net, [3, 3], scope=’pool’)

### TO GO FROM A CONVOLUTIONAL LAYER TO A FULLY CONNECTED YOU NEED TO FLATTEN THE ARRAY
net = slim.flatten(net, scope=’flat’)

###SLIM SPECIFYING A FULLY CONNECTED LAYER WHOSE OUT IS 512
net = slim.fully_connected(net, 512, scope=’fc_2’)

###SLIM SPECIFYING A FULLY CONNECTED LAYER WITHOUT ReLU WHOSE OUT IS 25
net = slim.fully_connected(net, 25, scope=’fc_3’, activation_fn=None)

return net

def get_acc(self, y_, y_out):
’’’
Fill in a way to compute accurracy given two tensorflows arrays
y_ (the true label) and y_out (the predict label)
’’’

cp = tf.equal(tf.argmax(y_out, 1), tf.argmax(y_, 1))

ac = tf.reduce_mean(tf.cast(cp, tf.float32))

return ac

def loss_layer(self, predicts, classes, scope=’loss_layer’):
’’’
The loss layer of the network, which is written for you.
You need to fill in get_accuracy to report the performance
’’’
with tf.variable_scope(scope):
self.class_loss = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(labels=classes, logits=predicts))

self.accurracy = self.get_acc(classes, predicts)
```

(b) The next step to train the network is to complete the pipeline which loads the datasets and offers it as mini-batches into the network. **Fill in the missing code in *data_manager.py* and report your code.**

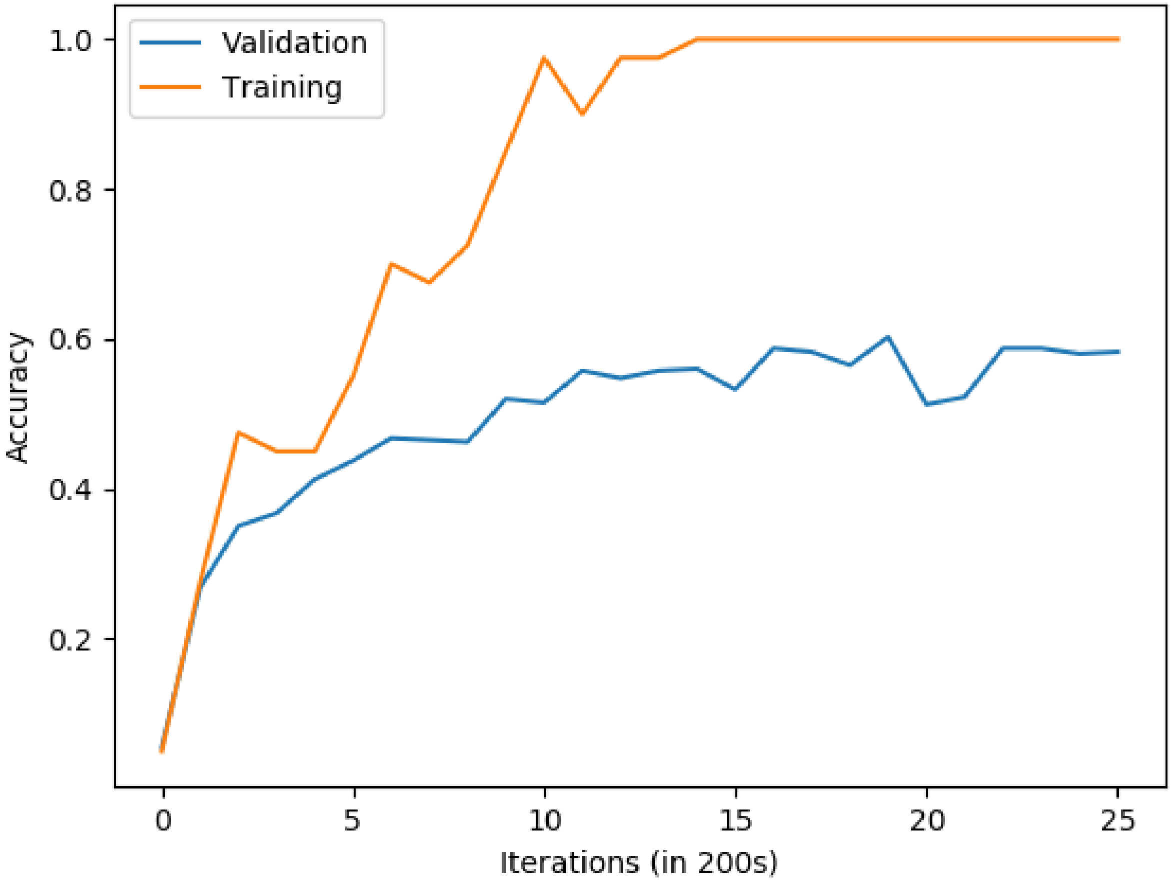
```
import copy
import glob
import os
import pickle

import IPython
import cv2
import numpy as np
from numpy.random import random


class data_manager(object):
def __init__(self, classes, image_size, compute_features=None, compute_label=None):

# Batch Size for training
self.batch_size = 40
# Batch size for test, more samples to increase accuracy
self.val_batch_size = 400

self.classes = classes
self.num_class = len(self.classes)
self.image_size = image_size

self.class_to_ind = dict(zip(self.classes, range(len(self.classes))))

# To keep track of where in the data you are in the training data
self.cursor = 0
# Same as above but for validation data
self.t_cursor = 0
self.epoch = 1

self.recent_batch = []

if compute_features == None:
self.compute_feature = self.compute_features_baseline

else:
self.compute_feature = compute_features

if compute_label == None:
self.compute_label = self.compute_label_baseline
else:
self.compute_label = compute_label

self.load_train_set()
self.load_validation_set()

def get_train_batch(self):

'''

Compute a training batch for the neural network
The batch size should be size 40

'''
images = []
```

```
labels = []

for i in range(self.batch_size):
images.append(self.train_data[self.cursor][’features’])
labels.append(self.train_data[self.cursor][’label’])

self.cursor += 1
if self.cursor == len(self.train_data):
np.random.shuffle(self.train_data)
self.cursor = 0

images = np.stack(images)
labels = np.stack(labels)

return images, labels

def get_empty_state(self):
images = np.zeros((self.batch_size, self.image_size, self.image_size, 3))
return images

def get_empty_label(self):
labels = np.zeros((self.batch_size, self.num_class))
return labels

def get_empty_state_val(self):
images = np.zeros((self.val_batch_size, self.image_size, self.image_size, 3))
return images

def get_empty_label_val(self):
labels = np.zeros((self.val_batch_size, self.num_class))
return labels

def get_validation_batch(self):

’’’
Compute a training batch for the neural network

The batch size should be size 400

’’’
# FILL IN
images = []
labels = []
for i in range(self.val_batch_size):

images.append(self.val_data[self.t_cursor][’features’])
labels.append(self.val_data[self.t_cursor][’label’])

self.t_cursor += 1
if self.t_cursor == len(self.val_data):
np.random.shuffle(self.val_data)
self.t_cursor = 0

images = np.stack(images)
labels = np.stack(labels)

return images, labels

def compute_features_baseline(self, image):
’’’
computes the featurized on the images. In this case this corresponds
to rescaling and standardizing.
’’’
```

```
image = cv2.resize(image, (self.image_size, self.image_size))
image = (image / 255.0) * 2.0 - 1.0

return image

def compute_label_baseline(self, label):
’’’
Compute one-hot labels given the class size
’’’

one_hot = np.zeros(self.num_class)

idx = self.classes.index(label)

one_hot[idx] = 1.0

return one_hot

def load_set(self, set_name):

’’’
Given a string which is either ’val’ or ’train’, the function should load all the
data into an

’’’

data = []
data_paths = glob.glob(set_name + ’/*.png’)

count = 0

for datum_path in data_paths:

label_idx = datum_path.find(’_’)


label = datum_path[len(set_name) + 1: label_idx]


if self.classes.count(label) > 0:
img = cv2.imread(datum_path)

label_vec = self.compute_label(label)

features = self.compute_feature(img)

data.append({’c_img’: img, ’label’: label_vec, ’features’: features})

np.random.shuffle(data)
return data

def load_train_set(self):
’’’
Loads the train set
’’’

self.train_data = self.load_set(r’D:\360Sync\Academia\HW13data\train’)

def load_validation_set(self):
’’’
Loads the validation set
’’’
```

```
self.val_data = self.load_set(r’D:\360Sync\Academia\HW13data\val’)
```

(c) We will now complete the iterative optimization loop. Fill in the missing code in *trainer.py* to iteratively apply SGD for a fix number of iterations. In our system, we will be using an extra momentum term to help speed up the SGD optimization. **Run the file *train_cnn.py* and report the resulting chart.**



```
import argparse
import datetime
import os
import sys

import tensorflow as tf

slim = tf.contrib.slim


class Solver(object):
def __init__(self, net, data):

self.net = net
self.data = data

# Number of iterations to train for
self.max_iter = 5000
# Every 200 iterations please record the trest and train loss
```

```
self.summary_iter = 200

’’’
Tensorflow is told to use a gradient descent optimizer
In the function optimize you will iteratively apply this on batches of data
’’’
self.train_step = tf.train.MomentumOptimizer(.003, .9)
self.train = self.train_step.minimize(self.net.class_loss)

self.sess = tf.Session()
self.sess.run(tf.global_variables_initializer())

def optimize(self):

# Append the current train and test accuracy every 200 iterations
self.train_accuracy = []
self.test_accuracy = []

’’’
Performs the training of the network.
Implement SGD using the data manager to compute the batches
Make sure to record the training and test accuracy through out the process
’’’
for i in range(self.max_iter + 1):
images, labels = self.data.get_train_batch()
self.sess.run(self.train, feed_dict={self.net.images: images, self.net.labels: labels})

if i % self.summary_iter == 0:
print(’i: ’ + str(i) + ’ is recored’)
accuracy = self.sess.run(self.net.accurracy,
feed_dict={self.net.images: images, self.net.labels: labels})
self.train_accuracy.append(accuracy)
images_val, labels_val = self.data.get_validation_batch()
val_accuracy = self.sess.run(self.net.accurracy,
feed_dict={self.net.images: images_val, self.net.labels: labels_val})
self.test_accuracy.append(val_accuracy)
```

(d) We have seen that the convolutional neural network can achieve a good performance on this dataset. However, we are not sure about whether a fully connected neural network could do the same job. We would like to replace the layers (a)-(c) in the original network by a fully connected layer and a ReLU layer. We want to have approximately the same number of parameters between layer (a) in the original network and the new fully connected layer. **How many hidden units will the new fully connected layer has? You can round up fractional numbers to the nearest integer. Do you observe any wierd design about this new network?**

Hint: For a convolution layer with input size $W \times H \times C$ and N convolution filters with each of them having a size of $X \times Y$, it has $C \times X \times Y \times N$ parameters.

Let's use the hint:
For a convolution layer with input size $W \times H \times C$ and N convolution filters with each of them having a size of $X \times Y$, it has $C \times X \times Y \times N$ parameters.
We have:

$$C \times X \times Y \times N = W \times H \times C \times U$$

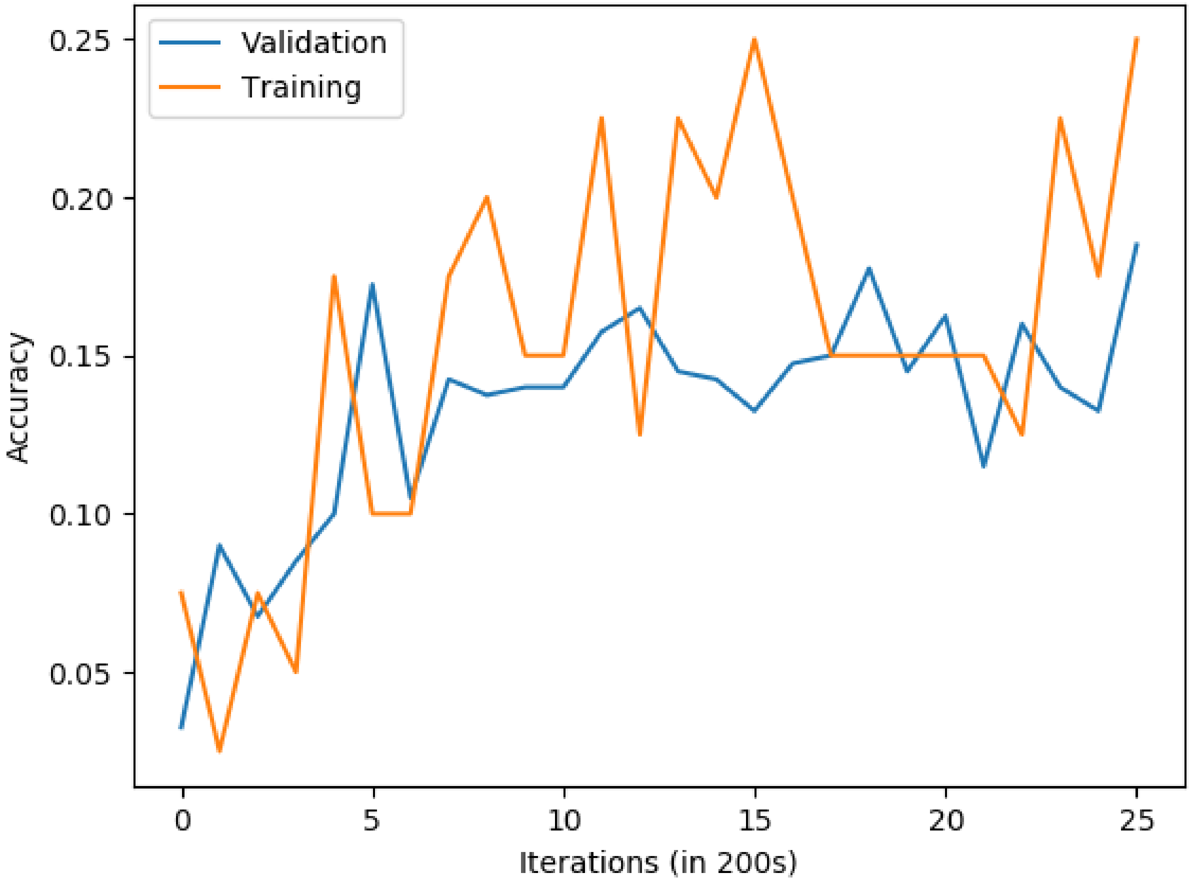
where U denotes the number of hidden units

$$U = \frac{X \times Y \times N}{W \times H}$$

For layer a, we will then have $15 * 15 * 5 * /90/90 \approx 1$ hidden unit. That's we have a extremely narrow layer a.

(e) Implement the fully connected network in part (d) and redo part (c). How's the performance of the fully connected network?

The performance of the fully connected network is awful. This is because it loses the local information.

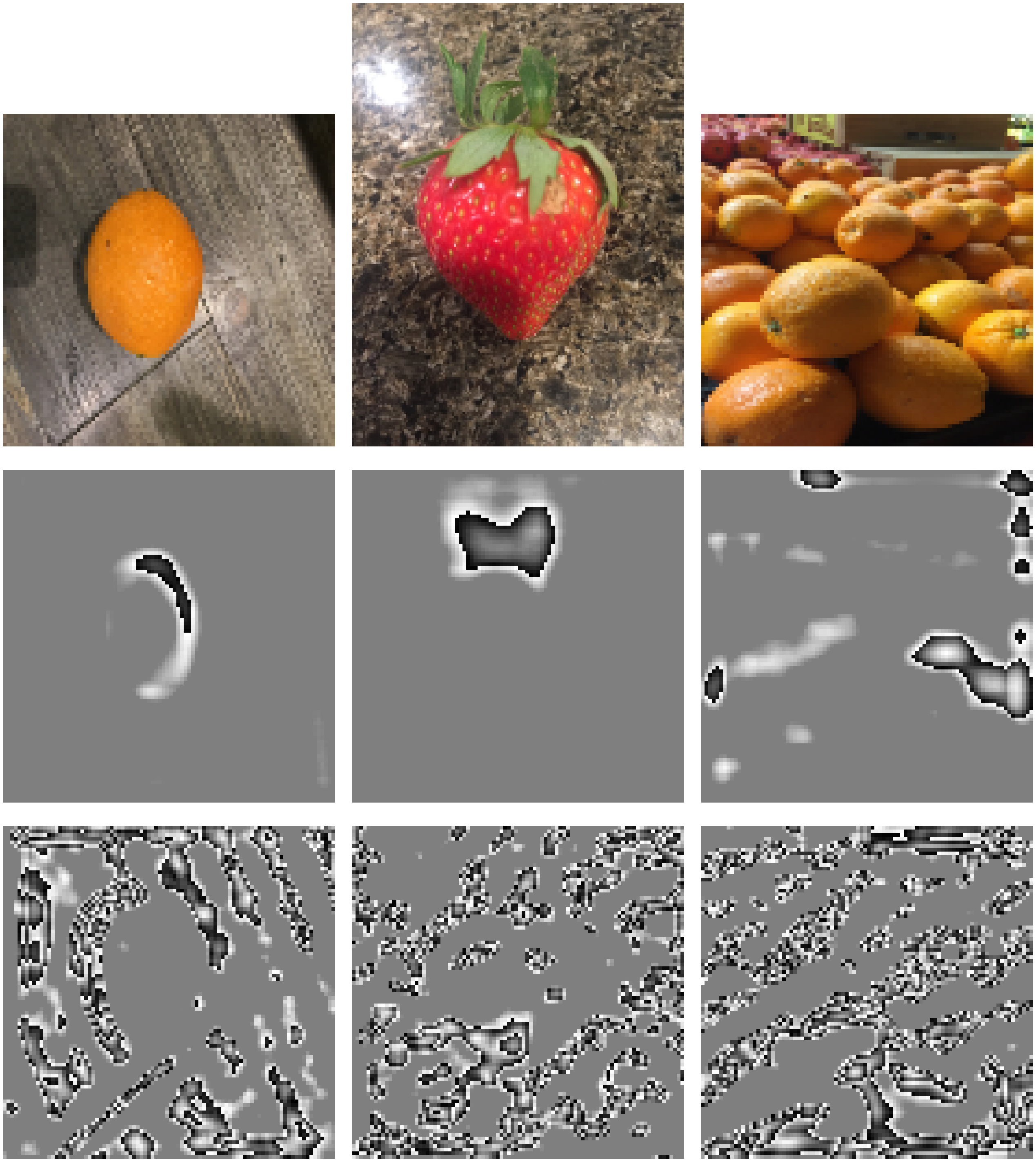


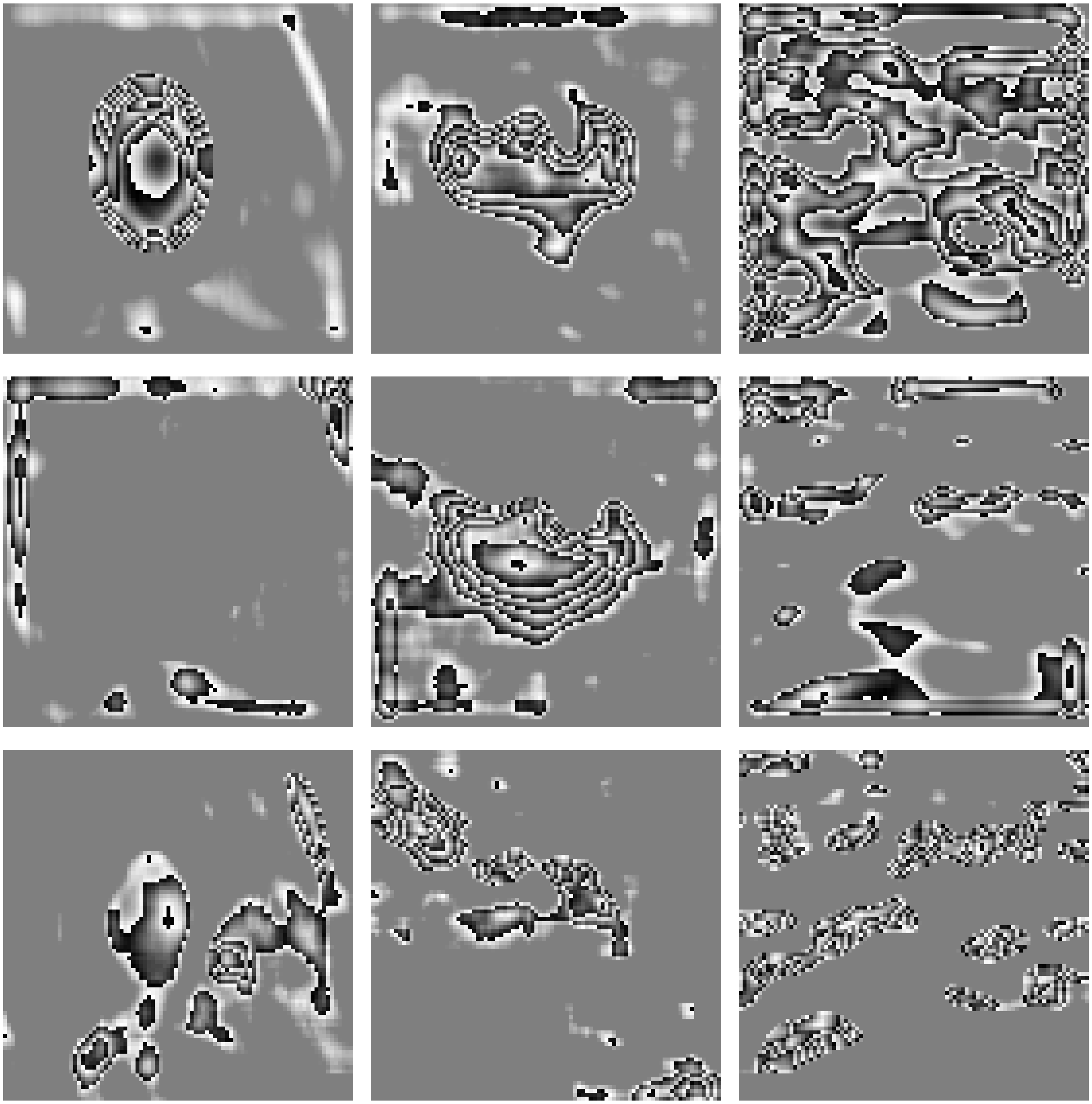
```
import tensorflow as tf

# import yolo.config_card as cfg

slim = tf.contrib.slim


class CNN(object):
def __init__(self, classes, image_size):
'''
Initializes the size of the network
'''

self.classes = classes
self.num_class = len(self.classes)
self.image_size = image_size

self.output_size = self.num_class
self.batch_size = 40
self.images = tf.placeholder(tf.float32, [None, self.image_size, self.image_size, 3], name=’images’)

self.logits = self.build_network(self.images, num_outputs=self.output_size)

self.labels = tf.placeholder(tf.float32, [None, self.num_class])

self.loss_layer(self.logits, self.labels)
self.total_loss = tf.losses.get_total_loss()
tf.summary.scalar(’total_loss’, self.total_loss)

def build_network(self,
images,
num_outputs,
scope=’yolo’):
with tf.variable_scope(scope):
with slim.arg_scope([slim.conv2d, slim.fully_connected],
weights_initializer=tf.truncated_normal_initializer(0.0, 0.01),
weights_regularizer=slim.l2_regularizer(0.0005)):
’’’
Fill in network architecutre here
Network should start out with the images function
Then it should return net
’’’

### TO GO FROM A CONVOLUTIONAL LAYER TO A FULLY CONNECTED YOU NEED TO FLATTEN THE ARRAY
net = slim.flatten(images, scope=’flat’)

###SLIM SPECIFYING A FULLY CONNECTED LAYER WHOSE OUT IS 1
net = slim.fully_connected(net, 1, scope=’fc_1’)

###SLIM SPECIFYING A FULLY CONNECTED LAYER WHOSE OUT IS 512
net = slim.fully_connected(net, 512, scope=’fc_2’)

###SLIM SPECIFYING A FULLY CONNECTED LAYER WITHOUT ReLU WHOSE OUT IS 25
net = slim.fully_connected(net, 25, scope=’fc_3’, activation_fn=None)

return net

def get_acc(self, y_, y_out):
’’’
Fill in a way to compute accurracy given two tensorflows arrays
y_ (the true label) and y_out (the predict label)
’’’

cp = tf.equal(tf.argmax(y_out, 1), tf.argmax(y_, 1))

ac = tf.reduce_mean(tf.cast(cp, tf.float32))

return ac

def loss_layer(self, predicts, classes, scope=’loss_layer’):
’’’
The loss layer of the network, which is written for you.
You need to fill in get_accuracy to report the performance
’’’
with tf.variable_scope(scope):
self.class_loss = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(labels=classes, logits=predicts))

self.accurracy = self.get_acc(classes, predicts)
```

```
from data_manager import data_manager
from cnn_fake import CNN
from trainer import Solver
from viz_features import Viz_Feat
import random


import matplotlib.pyplot as plt

CLASS_LABELS =
    [’apple’,’banana’,’nectarine’,’plum’,’peach’,’watermelon’,’pear’,’mango’,’grape’,’orange’,’strawberry’,’pineapple’,
’radish’,’carrot’,’potato’,’tomato’,’bellpepper’,’broccoli’,’cabbage’,’cauliflower’,’celery’,’eggplant’,’garlic’,’spina

LITTLE_CLASS_LABELS = [’apple’,’banana’,’eggplant’]

image_size = 90

random.seed(0)

classes = CLASS_LABELS
dm = data_manager(classes, image_size)

cnn = CNN(classes, image_size)

solver = Solver(cnn, dm)

solver.optimize()

plt.plot(solver.test_accuracy,label = ’Validation’)
plt.plot(solver.train_accuracy, label = ’Training’)
plt.legend()
plt.xlabel(’Iterations (in 200s)’)
plt.ylabel(’Accuracy’)
# plt.show()
plt.savefig(’Figure_3e.png’)

val_data = dm.val_data
train_data = dm.train_data

# sess = solver.sess

# cm = Viz_Feat(val_data,train_data,CLASS_LABELS,sess)

# cm.vizualize_features(cnn)
```

(f) To better understand, how the network was able to achieve the best performance on our fruits and veggies dataset. It is important to understand that it is learning features to reduce the dimensionality of the data. We can see what features were learned by examining the response maps after our convolutional layer.

The response map is the output image after the convolutional has been applied. This image can be interpreted as what features are interesting for classification. **Fill in the missing code in** *viz_features.py* **and report the images specified**.



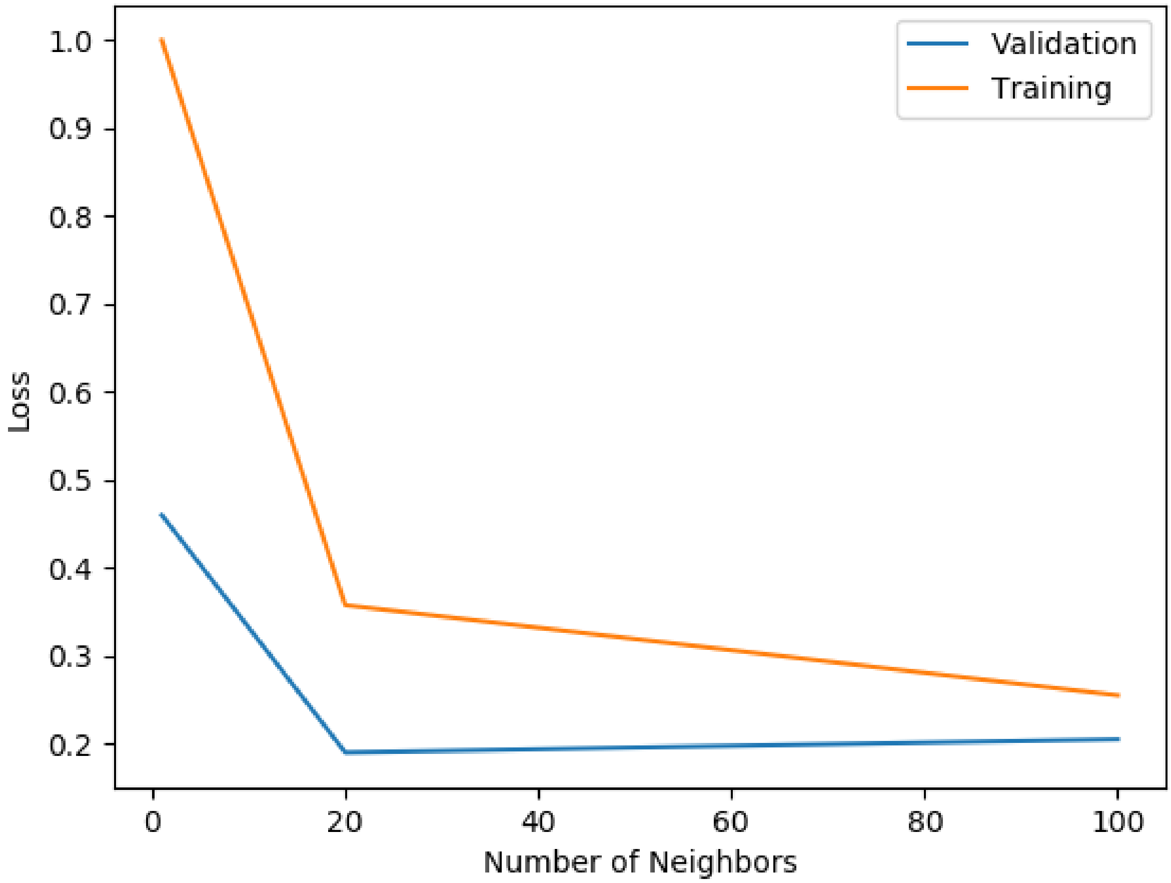


```
import random

import IPython
import cv2
import matplotlib.pyplot as plt
import numpy as np
from sklearn.metrics import confusion_matrix


class Viz_Feat(object):
def __init__(self, val_data, train_data, class_labels, sess):

self.val_data = val_data
self.train_data = train_data
self.CLASS_LABELS = class_labels
self.sess = sess

def vizualize_features(self, net):
```

```
images = [0, 10, 100]
’’’
Compute the response map for the index images
’’’
for i in images:

# validation data
curr_img = self.val_data[i][’features’]
curr_img = np.reshape(curr_img, (1,) + curr_img.shape)
curr_label = np.reshape(self.val_data[i][’label’], (1, -1))
response_map = self.sess.run(net.response_map,
feed_dict={net.images: curr_img, net.labels: curr_label})

class_label = str(self.CLASS_LABELS[np.nonzero(curr_label[0])[0][0]])
cv2.imwrite(’val_’ + str(i) + ’_’ + class_label + ’_raw.png’, self.val_data[i][’c_img’])

for ifilter in range(curr_img.shape[-1]):
cv2.imwrite(’val_’ + str(i) + ’_’ + class_label + ’_filter-’ + str(ifilter) + ’.png’,
self.revert_image(response_map[0, :, :, ifilter]))

def revert_image(self, img):
’’’
Used to revert images back to a form that can be easily visualized
’’’

img = (img + 1.0) / 2.0 * 255.0

img = np.array(img, dtype=int)

blank_img = np.zeros([img.shape[0], img.shape[1], 3])

blank_img[:, :, 0] = img
blank_img[:, :, 1] = img
blank_img[:, :, 2] = img

img = blank_img.astype("uint8")

return img
```

(g) Given that our network has achieved high generalization with such low training error, it suggests that a high variance estimator is appropriate for the task. To better understand why the network is able to work, we can compare it to another high variance estimator such as K-nearest neighbors. **Fill in the missing code in *nn_classifier.py* and report the performance as the numbers of neighbors is swept across when *train_nn.py* is run.**

It should be noted that loss actually means accuracy. The validation errors are terrible using kNN.



```
import glob
import os
import random
import sys
import time

import IPython
import matplotlib.pyplot as plt
import numpy as np
from numpy.random import uniform
from  sklearn.neighbors import KNeighborsClassifier


class NN():
def __init__(self, train_data, val_data, n_neighbors=5):
self.train_data = train_data
self.val_data = val_data

self.sample_size = 400

self.model = KNeighborsClassifier(n_neighbors=n_neighbors)

def preprocess(self, data):
X = [d['features'] for d in data]
y = np.array([d['label'] for d in data])

X = np.reshape(X, (len(data), -1))
y = np.argmax(y, axis=1)
return X, y

def train_model(self):
'''
Train Nearest Neighbors model
'''
X, y = self.preprocess(self.train_data)
self.model.fit(X, y)

def get_error(self, data):
X, y = self.preprocess(data)
indicies = np.array(np.random.choice(len(data), self.sample_size, replace=False), dtype=int)
X = X[indicies]
y = y[indicies]
yhat = self.model.predict(X)

return np.count_nonzero(y == yhat) / self.sample_size

def get_validation_error(self):
'''
Compute validation error. Please only compute the error on the sample_size number
over randomly selected data points. To save computation.
'''
return self.get_error(self.val_data)

def get_train_error(self):
'''
Compute train error. Please only compute the error on the sample_size number
over randomly selected data points. To save computation.
'''
return self.get_error(self.train_data)
```

Question 4. Running Time of k-Nearest Neighbour Search Methods

The method of k-nearest neighbours is a fundamental building block of machine learning algorithms. A classic example is the k-nearest neighbour classifier, which is a non-parametric classifier that finds the k closest examples in the training set to the test example and outputs the most common label among the k nearby examples as its prediction. Generating predictions using this classifier requires an algorithm to find the k closest examples in a possibly large and high-dimensional dataset, which is known as the k-nearest neighbour search problem. More precisely, given a set of n points, $\mathcal{D} = \{\vec{x}_1 \ldots, \vec{x}_n\} \subseteq \mathbb{R}^d$ and a query point $\vec{z} \in \mathbb{R}^d$, the problem requires finding k points in $\mathcal{D}$ that are the closest to $\vec{z}$ in Euclidean distance.

(a) First, we consider the naïve exhaustive search algorithm, which exhaustively computes the distance between $\vec{z}$ and all points in $\mathcal{D}$ and returns the k points with the shortest distance. **What is the (average case) time complexity for computing distances between the query and all points, finding the k shortest distances using quickselect? What is the (average case) time complexity of running the overall algorithm for a single query? Simplify the resulting expression and explain the reasoning behind the simplification.**

What is the (average case) time complexity for computing distances between the query and all points, finding the k shortest distances using quickselect?

First of all, I need to find a reference to quickselect. It appears that it has a time complexity of $\mathbb{O}(n)$.

On the other hand, computing distances between the query and all points has a time compleixty of $\mathbb{O}(nd)$.

What is the (average case) time complexity of running the overall algorithm for a single query? Simplify the resulting expression and explain the reasoning behind the simplification.

The overall time complexity is $\mathbb{O}(nd + n)$. Simplifying this gives us $\mathbb{O}(nd)$ because we assume $d > 1$ and nd dominates n.

(b) Decades of research have focused on devising a way of preprocessing the data so that the k-nearest neighbours for each query can be found efficiently. “Efficient” means the time complexity of finding the k-nearest neighbours, is lower than that of the naïve exhaustive search algorithm. Since complexity of exhaustive search scales linearly in n, the complexity of a more efficient algorithm must grow more slowly than linearly as n increases; in other words, its complexity must be sublinear in n.

Many efficient algorithms for k-nearest neighbour search rely on a divide-and-conquer strategy known as space partitioning. The idea is to divide the vector space into cells and maintain a data structure that keeps track of the points that lie in each. Then, to find the k-nearest neighbours of a query, these algorithms look up the cell that contains the query and obtain the subset of points in $\mathcal{D}$ that lie in the cell and adjacent cells. They then perform exhaustive search over this subset, i.e the distances from each point in the subset and the query are computed and the k points in the subset that are the closest to the query are returned.

For simplicity, we’ll consider the special case of $k = 1$ in the following questions, but note that the various algorithms we’ll consider can be easily extended to the setting with arbitrary k. We first consider a simple partitioning scheme, where we place a Cartesian grid (a rectangular grid consisting of hypercubes) over the vector space.

Figure 1: Illustration of the space partitioning scheme we consider. The data points are shown as blue circles and the query is shown as the red square. The cell boundaries are shown as gold lines.

How many cells need to be searched in total if the data points are one-dimensional? Two-dimensional? d-dimensional? If each cell contains one data point, what is the time complexity for finding the 1-nearest neighbour in terms of d, assuming accessing any cell takes constant time?

We need to search 3 cells for one-dimensional.
We need to search 9 cells for two-dimensional.
We need to search 3^d cells for d-dimensional.
Computing the distance requires $\mathbb{O}(d)$ time. Therefore, the time complexity for finding the 1-nearest neightbour is $\mathbb{O}(d3^d)$.

(c) In low dimensions, 3^d is much less than n, and so this method provides a significant speedup over naïve exhaustive search. However, in moderately high dimensions, because time complexity is exponential in dimensionality, 3^d can easily exceed n; in this case, the number of points retrieved from neighbouring cells is simply n, and so this method would not provide any speedup over exhaustive search. So, if we account for the total number of points in the dataset, the query time complexity is $O(d\ \min(3^d, n))$. This exponential dependence on d arises in many settings, and is often known as *the curse of dimensionality.*

How do we overcome the curse of dimensionality? Since it arises from the need to search adjacent cells, what if we don't have cells at all?

Consider a new approach that simply projects all data points along a uniformly randomly chosen direction and keeps all projections of data points in a sorted list. To find the 1-nearest neighbour, the algorithm projects the query along the same direction used to project the data points and uses binary search to find the data point whose projection is closest to that of the query. Then it marches along the list to obtain $\tilde{k}$ points whose projections are the closest to the projection of the query. Finally, it performs exhaustive search over these points and returns the point that is the closest to the query. This is a simplified version of an algorithm known as Dynamic Continuous Indexing (DCI).

Because this algorithm is randomized (since it uses a randomly chosen direction), there is a non-zero probability that it returns the incorrect results. We are therefore interested in how many points we need to exhaustively search over to ensure the algorithm succeeds with high probability. Equivalently, we'd like to upper bound the failure probability in terms of the number of points to search over.

The probability we consider first is the probability that a data point that is originally far away appears closer to the query under projection than a data point that is originally close. We can assume without loss of generality that the query is at the origin. Hence this probability is the same as the probability of a long vector appearing shorter than a short vector under projection. Let $\vec{v}^l \in \mathbb{R}^d$ and $\vec{v}^s \in \mathbb{R}^d$ denote the long and short vectors respectively and $\vec{u} \in S^{d-1} \subset \mathbb{R}^d$ denote a vector drawn uniformly randomly on the unit sphere.

Assuming that $\vec{v}^l$ and $\vec{v}^s$ are not collinear, consider the plane spanned by $\vec{v}^l$ and $\vec{v}^s$, which we will denote as P. (If $\vec{v}^l$ and $\vec{v}^s$ are collinear, we define P to be the subspace spanned by $\vec{v}^l$ and an arbitrary vector that's linearly independent of $\vec{v}^l$.) For any vector $\vec{w}$, we use $\vec{w}^{\parallel}$ and $\vec{w}^{\perp}$ to denote the components of $\vec{w}$ in P and $P^{\perp}$ such that $\vec{w} = \vec{w}^{\parallel} + \vec{w}^{\perp}$.

For $\vec{w} \in \{\vec{v}^s, \vec{v}^l\}$, because $\vec{w}^{\perp} = 0$, $\langle \vec{w}, \vec{u} \rangle = \langle \vec{w}, \vec{u}^{\parallel} \rangle$. So, $\Pr\left(\left|\langle \vec{v}^l, \vec{u} \rangle\right| \leq \left|\langle \vec{v}^s, \vec{u} \rangle\right|\right) = \Pr\left(\left|\langle \vec{v}^l, \vec{u}^{\parallel} \rangle\right| \leq \left|\langle \vec{v}^s, \vec{u}^{\parallel} \rangle\right|\right)$. **If we use θ denote the angle of $\vec{u}^{\parallel}$ relative to $\vec{v}^l$, show that this quantity is at most $\Pr\left(|\cos\theta| \leq \left\|\vec{v}^s\right\|_2 / \left\|\vec{v}^l\right\|_2\right)$.**

Figure 2: Examples of "good" and "bad" projection directions. The plot shows the plane P spanned by $\vec{v}^l$ and $\vec{v}^s$. The vectors $\vec{v}^l$ and $\vec{v}^s$ are shown as black arrows. The blue lines denote possible projection directions $\vec{u}^{\parallel}$. The isolated blue line represents a "good" projection direction, since the projection of $\vec{v}^l$ is longer than the projection of $\vec{v}^s$ (both shown in green), thereby preserving the relative order between $\vec{v}^l$ and $\vec{v}^s$ in terms of their lengths after projection. Any projection direction within the shaded region is a "bad" projection direction, since the projection of $\vec{v}^l$ would not be longer than the projection of $\vec{v}^s$, thereby inverting the relative order between $\vec{v}^l$ and $\vec{v}^s$ after projection. Two examples of such projection directions are shown, which lie on the boundaries of the shaded region. Along each of these directions, the projection $\vec{v}^l$ and the projection of $\vec{v}^s$ are of equal length (both of which are shown in red). The area of the shaded region represents the order-inversion probability.

$\Pr\left(\left|\langle \vec{v}^l, \vec{u} \rangle\right| \leq \left|\langle \vec{v}^s, \vec{u} \rangle\right|\right)$

$=\Pr\left(\left|\langle \vec{v}^l, \vec{u}^{\|}\rangle\right| \leq \left|\langle \vec{v}^s, \vec{u}^{\|}\rangle\right|\right)$

$=\Pr\left(\left|\|\vec{v}^l\|_2\|\vec{u}^{\|}\|_2 \cos\theta\right| \leq \left|\|\vec{v}^s\|_2\|\vec{u}^{\|}\|_2 \cos\psi\right|\right)$

ψ denotes the angle of $\vec{u}^{\|}$ relative to $\vec{v}^s$, we know that $\cos\psi \leq 1$

now we can get the upper bound of the probability by using the upper bound of the inequality

This is due to the continuous distribution of ϕ

$\leq\Pr\left(\left|\|\vec{v}^l\|_2\|\vec{u}^{\|}\|_2 \cos\theta\right| \leq \left|\|\vec{v}^s\|_2\|\vec{u}^{\|}\|_2\right|\right)$

$=\Pr\left(|\cos\theta| \leq \left\|\vec{v}^s\right\|_2 / \left\|\vec{v}^l\right\|_2\right)$

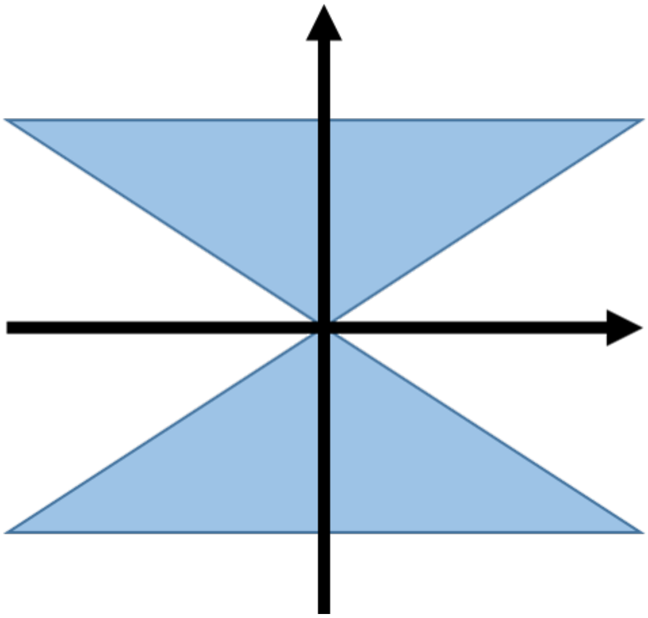
(d) **Derive the range of θ such that $|\cos\theta| \leq \left\|\vec{v}^s\right\|_2 / \left\|\vec{v}^l\right\|_2$ and show that $\Pr\left(|\cos\theta| \leq \left\|\vec{v}^s\right\|_2 / \left\|\vec{v}^l\right\|_2\right) = 1 - \frac{2}{\pi}\cos^{-1}\left(\left\|\vec{v}^s\right\|_2 / \left\|\vec{v}^l\right\|_2\right)$.**

Hint: Due to rotational invariance of a uniform distribution on the sphere, the angle between $\vec{u}^{\|}$ and any vector in P is uniformly distributed.

Derive the range of θ:

$$\begin{aligned} |\cos\theta| &\leq \left\|\vec{v}^s\right\|_2 / \left\|\vec{v}^l\right\|_2 \\ -\left\|\vec{v}^s\right\|_2 / \left\|\vec{v}^l\right\|_2 \leq \cos\theta &\leq \left\|\vec{v}^s\right\|_2 / \left\|\vec{v}^l\right\|_2 \\ \cos^{-1}\left(\left\|\vec{v}^s\right\|_2 / \left\|\vec{v}^l\right\|_2\right) + m\pi \leq \theta &\leq \cos^{-1}\left(-\left\|\vec{v}^s\right\|_2 / \left\|\vec{v}^l\right\|_2\right) + (m+1)\pi \\ \cos^{-1}\left(\left\|\vec{v}^s\right\|_2 / \left\|\vec{v}^l\right\|_2\right) + m\pi \leq \theta &\leq \pi - \cos^{-1}\left(\left\|\vec{v}^s\right\|_2 / \left\|\vec{v}^l\right\|_2\right) + m\pi \end{aligned}$$

Let's look at the possible θ shown by the blue region.



That is the probability is computed by the fraction of blue region to the entire subspace.

$$\begin{aligned} &\Pr\left(|\cos\theta| \leq \left\|\vec{v}^s\right\|_2 / \left\|\vec{v}^l\right\|_2\right) \\ =&\frac{2\left(\pi - 2\cos^{-1}\left(\left\|\vec{v}^s\right\|_2 / \left\|\vec{v}^l\right\|_2\right)\right)}{2\pi} \\ =&1 - \frac{2}{\pi}\cos^{-1}\left(\left\|\vec{v}^s\right\|_2 / \left\|\vec{v}^l\right\|_2\right) \end{aligned}$$

(e) The part above shows that the relative ordering of two data points is more likely to flip if their distances to the query are not very different. So, intuitively, the nearest neighbour search problem is harder if all data points are almost equidistant from the query. More concretely, we can consider a dataset consisting of many points on a sphere and a single point placed at a location we will choose later. For a query placed at the centre of the sphere, the 1-nearest neighbour search problem is easy if the single data point is placed near the centre, since it is much closer to the query than the other data points. On the other hand, if it is placed near the surface of the ball but still inside the ball, the problem is much harder since the single data point is only slightly closer to the query than the other data points and so is a much closer call. Therefore, the intrinsic hardness of the problem is characterized by the distribution of distances to the query.

The algorithm would fail to return the correct set of 1-nearest neighbour if more than $\tilde{k}-1$ points appear closer to the query than the 1-nearest neighbour under projection. The following lemma is useful:

> For any set of events $\{E_i\}_{i=1}^N$, the probability that at least k' of them occur is at most $\frac{1}{k'}\sum_{i=1}^N \Pr(E_i)$.

This is a generalization of the union bound; the lemma reduces to the union bound when $k'=1$. (See this paper [1] for the proof – let us know if you can come up with a simpler proof :)). **Using the lemma, derive an upper bound on the probability that the algorithm fails, which is known as the *failure probability*. Use $\vec{x}^{(i)}$ to denote the ith closest point to the query $\vec{z}$. Then use the fact that $1-\frac{2}{\pi}\cos^{-1}(t) \le t$ for all $t\in[0,1]$ to simplify the expression.**

Hint:

$$\Pr(\text{algorithm fails}) = \Pr\left(\text{at least } \tilde{k} \text{ points are closer to } \vec{z} \text{ than } \vec{x}^{(1)} \text{ under projection } \vec{u}\right)$$

$$\Pr(\text{algorithm fails})$$

$$=\Pr\left(\text{at least } \tilde{k} \text{ points are closer to } \vec{z} \text{ than } \vec{x}^{(1)} \text{ under projection } \vec{u}\right)$$

now we have a set of events where E_i means $\vec{x}^{(i)}$ is closer than $\vec{x}^{(1)}$

$$=\frac{1}{\tilde{k}}\sum_{i=2}^{N}\Pr(E_i)$$

$$=\frac{1}{\tilde{k}}\sum_{i=2}^{N}\left[1-\frac{2}{\pi}\cos^{-1}\left(\left\|\vec{v}^1\right\|_2/\left\|\vec{v}^i\right\|_2\right)\right]$$

Now we use the fact that $1-\dfrac{2}{\pi}\cos^{-1}(t)\le t$ for all $t\in[0,1]$ to simplify the expression

$$\le\frac{1}{\tilde{k}}\sum_{i=2}^{N}\left(\left\|\vec{v}^1\right\|_2/\left\|\vec{v}^i\right\|_2\right)$$

It should be noted that we assume the query point is at the origin. If not, we can change the expression to the one in the next part.

$$\Pr(\text{algorithm fails}) \le \frac{1}{\tilde{k}}\sum_{i=2}^{n}\left\|\vec{x}^{(1)}-\vec{z}\right\|_2/\left\|\vec{x}^{(i)}-\vec{z}\right\|_2$$

[1] Ke Li and Jitendra Malik. Fast k-Nearest Neighbour Search via Prioritized DCI. In *Proceedings of the 34th International Conference on Machine Learning*, pages 2081–2090, 2017.

(f) Notice that the failure probability does not depend on dimensionality at all. It only depends on the distribution of distances from every point to the query, which if we recall, measures the intrinsic hardness of the problem.

What's a typical distribution of distances? Natural data usually lies on a manifold, which is a generalization of Euclidean subspace that can be "curved" (more concretely, there is a neighbourhood around every point on the manifold that resembles a low-dimensional Euclidean space). For simplicity, we'll consider the case when the data is uniformly distributed on a d'-dimensional subspace, where d' is much less than the ambient dimensionality d. Often, d' is known as the intrinsic dimensionality. Then the number of points inside a ball of radius r is roughly $cr^{d'}$ for some constant c. So, the number of points inside a ball of constant radius grows exponentially in d'.

Assume for all r such that $cr^{d'}$ is an integer, the number of points inside a ball centred at $\vec{z}$ of radius r is exactly $cr^{d'}$. This is equivalent to saying $\left\|\vec{x}^{(cr^{d'})} - \vec{z}\right\|_2 = r$ for any such r. (If we recall from the previous part, $\vec{x}^{(i)}$ denotes the ith closest point to $\vec{z}$.) **Show the quantity $\sum_{i=2}^{n} \left\|\vec{x}^{(1)} - \vec{z}\right\|_2 / \left\|\vec{x}^{(i)} - \vec{z}\right\|_2$ in this case is $\sum_{i=2}^{n} (1/i)^{1/d'}$.**

Hint: to derive an expression for $\left\|\vec{x}^{(i)} - \vec{z}\right\|_2$ in terms of i, substitute i for $cr^{d'}$ in the equality $\left\|\vec{x}^{(cr^{d'})} - \vec{z}\right\|_2 = r$.

From the equality given in the question we have:

$$\left\|\vec{x}^{(cr^{d'})} - \vec{z}\right\|_2 = r$$

$$\left\|\vec{x}^{i} - \vec{z}\right\|_2 = \left(\frac{i}{c}\right)^{1/d'}$$

$$\sum_{i=2}^{n} \left\|\vec{x}^{(1)} - \vec{z}\right\|_2 / \left\|\vec{x}^{(i)} - \vec{z}\right\|_2$$

$$= \sum_{i=2}^{n} \left[\left(\frac{1}{c}\right)^{1/d'} / \left(\frac{i}{c}\right)^{1/d'}\right]$$

$$= \sum_{i=2}^{n} (1/i)^{1/d'}$$

(g) (BONUS) **Show the quantity $\sum_{i=2}^{n} (1/i)^{1/d'}$ is less than $\left(n^{1-1/d'} - 1\right) / (1 - 1/d')$.**

Hint: use the fact that $\sum_{i=a}^{b} \phi(i) = \int_{a}^{b+1} \phi(\lfloor t \rfloor) dt$ for any function ϕ and $t - 1 < \lfloor t \rfloor$ for any t, where $\lfloor \cdot \rfloor$ denotes the floor operator.

$$\begin{aligned}
&\sum_{i=2}^{n} (1/i)^{1/d'} \\
&= \int_{2}^{n+1} (1/\lfloor t \rfloor)^{1/d'} dt \\
&\leq \int_{2}^{n+1} (1/(t-1))^{1/d'} dt \\
&= \frac{1}{1-1/d'} (t-1)^{1-1/d'} \Big|_{2}^{n+1} \\
&= \left(n^{1-1/d'} - 1\right) / (1 - 1/d')
\end{aligned}$$

(h) (BONUS) **Show the failure probability is at most $O\left(n^{1-1/d'}/\tilde{k}\right)$ for $d' \geq 2$.**

$$
\begin{aligned}
&\Pr\left(\text{algorithm fails}\right)\\
\leq&\frac{1}{\tilde{k}}\sum_{i=2}^{n}\left\|\vec{x}^{(1)}-\vec{z}\right\|_2/\left\|\vec{x}^{(i)}-\vec{z}\right\|_2\\
=&\frac{1}{\tilde{k}}\sum_{i=2}^{n}\left(1/i\right)^{1/d'}\\
\sim&\frac{1}{\tilde{k}}\left(n^{1-1/d'}-1\right)/\left(1-1/d'\right)\\
&\text{when } d' \text{ is big, the denominator of the second fraction gets close to one}\\
&\text{the } -1 \text{ is also neglectable because we are dealing with } \mathbb{O}\\
\sim&\mathbb{O}\left(n^{1-1/d'}/\tilde{k}\right)
\end{aligned}
$$

(i) If we choose the number of points to search over, $\tilde{k}$, to be $\Omega\left(n^{1-1/d'}\right)$, we can ensure that the failure probability is strictly less than 1. Choosing this value for $\tilde{k}$ would mean that the query time complexity is $O(d(n^{1-1/d'}))$.

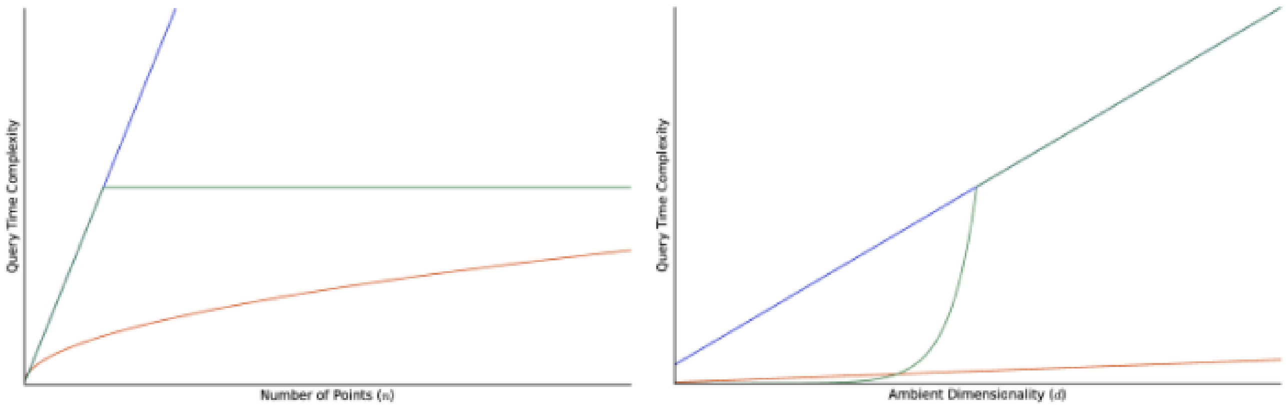
Consider the following variant of the algorithm, which essentially repeats the algorithm L times. More concretely, the algorithm projects the data points along L independently chosen random directions and maintain L sorted lists of projections, each corresponding to one projection direction. Given a query, we find the $\tilde{k}$ closest points to the query along each of the L directions and exhaustively search over the union of these points.

If we use α denote the failure probability of the original algorithm, the failure probability of this algorithm is at most α^L, because the projection direction are independently chosen. The query time complexity is $O(Ld(n^{1-1/d'}))$. Therefore, we can get an exponential decrease in the failure probability at a cost of a linear increase in the query time complexity.

We can choose a large enough L to make the failure probability arbitrarily small and therefore make the algorithm succeed with arbitrarily high probability. By convention, L is viewed as a constant (rather than a function of the failure probability), and so the time complexity for this algorithm is $O(dn^{1-1/d'})$. In general, this is how the time complexity of a randomized algorithm is derived: the parameters of the algorithm are chosen so that the algorithm can succeed with arbitrarily high probability, and time complexity is computed for that choice of parameters.

Observe that the time complexity has a linear dependence on d, which is better than the exponential dependence on d of the space partitioning-based approach. In addition, it has a sublinear dependence on d', which is surprising, since the number of points inside a ball of constant radius is exponential in d'.

The following plots show the query time complexities of naïve exhaustive search, space partitioning and DCI as functions of n and d. Curves of the same colour correspond to the same algorithm. **Which algorithm does each colour correspond to?**



Let's list the time complexities of all three algorithms here:

1. naïve search $\mathbb{O}(dn)$
2. cell search combined with naïve search (mixed cell search) $\mathbb{O}(d\min(3^d, n))$
3. random projection $\mathbb{O}\left(n^{1-1/d'}/\tilde{k}\right)$

The naïve search increases with the number of data points without a cap. Therefore, the blue curve is the naïve search.
The mixed cell search has a cap for the increase with the number of data points so the green curve is the mixed cell search.
The random projection has a much slower increase without a cap which fits with the read curve.
We can confirm this by looking at their behaviors with ambient dimensionality.
Again, the naïve search is supposed to be linear.
the mixed cell search as a kink and merge to the naïve search after d' gets too large.
the random projection increases much slower as dimension increases.

1. naïve search is blue
2. cell search combined with naïve search (mixed cell search) is green
3. random projection is red

Question 5. Your Own Question

Write your own question, and provide a thorough solution.

Writing your own problems is a very important way to really learn the material. The famous "Bloom's Taxonomy" that lists the levels of learning is: Remember, Understand, Apply, Analyze, Evaluate, and Create. Using what you know to create is the top-level. We rarely ask you any HW questions about the lowest level of straight-up remembering, expecting you to be able to do that yourself. (e.g. make yourself flashcards) But we don't want the same to be true about the highest level.

As a practical matter, having some practice at trying to create problems helps you study for exams much better than simply counting on solving existing practice problems. This is because thinking about how to create an interesting problem forces you to really look at the material from the perspective of those who are going to create the exams.

Besides, this is fun. If you want to make a boring problem, go ahead. That is your prerogative. But it is more fun to really engage with the material, discover something interesting, and then come up with a problem that walks others down a journey that lets them share your discovery. You don't have to achieve this every week. But unless you try every week, it probably won't happen ever.

What are the KKT conditions for the optimization problem below?

$$\min_x f(x) \quad s.t. g(x) \leq 0$$

The answer is:

1. Stationary: $\frac{dL}{dx} = \frac{df(x)}{dx} + \alpha^\top \frac{dg(x)}{dx} = 0$

2. Primal feasibility:$g(x) \leq 0$

3. Dual feasibility: $\alpha \geq 0$

4. Complementary slackness: $\alpha_i g(x)_i = 0$