CS480/680: Introduction to Machine Learning

Homework 2

Due: 11:59 pm, June 17, 2024, submit on LEARN.

NAME

student number

Submit your writeup in pdf and all source code in a zip file (with proper documentation). Write a script for each programming exercise so that the TA can easily run and verify your results. Make sure your code runs!
[Text in square brackets are hints that can be ignored.]

Exercise 1: Graph Kernels (6 pts)

One cool way to construct a new kernel from an existing set of (base) kernels is through graphs. Let $\mathcal{G} = (V, E)$ be a directed acyclic graph (DAG), where V denotes the nodes and E denotes the arcs (directed edges). For convenience let us assume there is a source node s that has no incoming arc and there is a sink node t that has no outgoing arc. We put a base kernel κ_e (that is, a function $\kappa_e : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$) on each arc $e = (u \to v) \in E$. For each path $P = (u_0 \to u_1 \to \cdots \to u_d)$ with $u_{i-1} \to u_i$ being an arc in E, we can define the kernel for the path P as the product of kernels along the path:

$$\forall \mathbf{x}, \mathbf{z} \in \mathcal{X},\ \kappa_P(\mathbf{x}, \mathbf{z}) = \prod_{i=1}^{d} \kappa_{u_{i-1} \to u_i}(\mathbf{x}, \mathbf{z}). \tag{1}$$

Then, we define the kernel for the graph $\mathcal{G}$ as the sum of all possible $s \to t$ *path kernels*:

$$\forall \mathbf{x}, \mathbf{z} \in \mathcal{X},\ \kappa_{\mathcal{G}}(\mathbf{x}, \mathbf{z}) = \sum_{P \in \text{path}(s \to t)} \kappa_P(\mathbf{x}, \mathbf{z}). \tag{2}$$

1. (1 pt) Prove that $\kappa_{\mathcal{G}}$ is indeed a kernel. [You may use any property that we learned in class about kernels.]

   Ans:
   See Slide 11 in Lecture 07 for Kernel Calculus.
   We start by stating for any path P, $\kappa_P(\mathbf{x}, \mathbf{z})$ is a kernel as products of kernels are also kernels.
   Then $\kappa_{\mathcal{G}}(\mathbf{x}, \mathbf{z})$ is also a kernel as sums of kernels are also kernels.

2. (2 pts) Let $\kappa_i, i = 1, \ldots, n$ be a set of given kernels. Construct a graph $\mathcal{G}$ (with appropriate base kernels) so that the graph kernel $\kappa_{\mathcal{G}} = \sum_{i=1}^{n} \kappa_i$. Similarly, construct a graph $\mathcal{G}$ (with appropriate base kernels) so that the graph kernel $\kappa_{\mathcal{G}} = \prod_{i=1}^{n} \kappa_i$.

   Ans:
   For our first graph, we introduce n intermediate nodes such that arc $(s, i) = \kappa_i$ and arc $(i, t) = 1$ (identity mapping). Then the resulting graph kernel is the summation of $\kappa_i \times 1 = \kappa_i$ as desired.

$$\begin{tikzcd}
 & 1 \arrow[rdd, "1"] & \\
 & 2 \arrow[rd, "1"] & \\
s \arrow[ruu, "\kappa_1"] \arrow[ru, "\kappa_2"] \arrow[rd, "\cdots"] \arrow[rdd, "\kappa_n"'] & & t \\
 & \cdots \arrow[ru, "\cdots"] & \\
 & n \arrow[ruu, "1"'] &
\end{tikzcd}$$

For our second graph, we connect n intermediate nodes together such that we have one directed path from s to t. Then we find our path kernel as $\kappa_P = \prod_{i=1}^n \kappa_i$ and since this is our only path, $\kappa_G = \prod_{i=1}^n \kappa_i$ as desired.

$$\begin{tikzcd}
s \arrow[r, "\kappa_1"] & 1 \arrow[r, "\kappa_2"] & 2 \arrow[r, "\ldots"] & \ldots \arrow[r, "\kappa_n"] & n \arrow[r, "1"] & t
\end{tikzcd}$$

3. (1 pt) Consider the subgragh of the figure below that includes nodes s, a, b, c (and arcs connecting them). Compute the graph kernel where s and c play the role of source and sink, respectively. Repeat the computation with the subgraph that includes s, a, b, c, d (and arcs connecting them), where d is the sink now.

$$\begin{tikzcd}
s \arrow[r, "xz"] & a \arrow[r, "e^{-(x-z)^2}"'] \arrow[rr, bend left, "(xz-1)^2"] & b \arrow[rr, bend right, "\tanh(xz+1)"'] & c \arrow[r, "e^{-|x-z|/2}"] & d \arrow[r, "1"] & t
\end{tikzcd}$$

Ans: For $(s-c)$, we note that there is one path connecting them: $P = s-a-c$. Then we find $\kappa_P = xz(xz-1)^2$ and since this is our only path $\kappa_G = xz(xz-1)^2$.

For $(s-d)$, we note that there are two paths connecting them $P_1 = s-a-c-d$ and $P_2 = s-a-b-d$. Then we find $\kappa_{P_1} = xz(xz-1)^2 e^{-|x-z|/2}$ and $\kappa_{P_2} = xze^{-(x-z)^2}\tanh(xz+1)$.
Then our graph kernel is the sum of these two path kernels:
$\kappa_G = xz(xz-1)^2 e^{-|x-z|/2} + xze^{-(x-z)^2}\tanh(xz+1)$.

4. (2 pts) Find an efficient algorithm to compute the graph kernel $\kappa_{\mathcal{G}}(\mathbf{x}, \mathbf{z})$ (for two fixed inputs $\mathbf{x}$ and $\mathbf{z}$) in time $O(|V|+|E|)$, assuming each base kernel κ_e costs $O(1)$ to evaluate. You may assume there is always at least one $s-t$ path. State and justify your algorithm is enough; no need (although you are encouraged) to give a full pseudocode.

[Note that the total number of paths in a DAG can be exponential in terms of the number of nodes $|V|$, so naive enumeration would not work. For example, replicating the intermediate nodes in the above figure n times creates 2^n paths from s to t.]

[Hint: Recall that we can use topological sorting to rearrange the nodes in a DAG such that all arcs go from a "smaller" node to a "bigger" one.]

Ans:
Our algorithm works in the following steps:

1. Topologically sort vertices.
2. Create **kernel** list of size $|V|$ where $\textbf{kernel}[s] = 1$ and 0 otherwise.
3. For every vertex v in topological sort:
3a. For every child of v, x, $\textbf{kernel}[x] += \textbf{kernel}[v] * \kappa_{(v,x)}$.
4. Return $\kappa_G = \textbf{kernel}[t]$.

This algorithm works by considering the vertices of G in-order. That is, for every node we visit, we are guaranteed to visit each of its parents (as pre defintion of topologically sorting DAG).
So we generate κ_G per node x as if it was the sink node. We do this by having considered all paths to x before hand, say to each parent v, and adding (the kernel where we pretended v was the sink node) $\times\ \kappa_{(v,x)}$ to κ_G. We add to κ_G to consider each such v. If we iterate in this way, by the tme we reach t, we have found κ_G where t is the sink node.

The runtime of this algorithm is $O(|V|+|E|)$ as topologically sorting vertices takes $O(|V|+|E|)$ time, and we visit each node exactly once and each edge exactly once.

Exercise 2: CNN Implementation (8 pts)

Note: Please mention your Python version (and maybe the version of all other packages).

In this exercise you are going to run some experiments involving CNNs. You need to know Python and install the following libraries: Pytorch, matplotlib and all their dependencies. You can find detailed instructions and tutorials for each of these libraries on the respective websites.

For all experiments, running on CPU is sufficient. You do not need to run the code on GPUs, although you could, using for instance Google Colab. Before start, we suggest you review what we learned about each layer in CNN, and read at least this tutorial.

1. Implement and train a VGG11 net on the MNIST dataset. VGG11 was an earlier version of VGG16 and can be found as model A in Table 1 of this paper, whose Section 2.1 also gives you all the details about each layer. The goal is to get the loss as close to 0 as possible. Note that our input dimension is different from the VGG paper. You need to resize each image in MNIST from its original size 28×28 to 32×32 [why?].

   For your convenience, we list the details of the VGG11 architecture here. The convolutional layers are denoted as `Conv(number of input channels, number of output channels, kernel size, stride, padding)`; the batch normalization layers are denoted as `BatchNorm(number of channels)`; the max-pooling layers are denoted as `MaxPool(kernel size, stride)`; the fully-connected layers are denoted as `FC(number of input features, number of output features)`; the drop out layers are denoted as `Dropout(dropout ratio)`:

   ```
   - Conv(001, 064, 3, 1, 1) - BatchNorm(064) - ReLU - MaxPool(2, 2)
   - Conv(064, 128, 3, 1, 1) - BatchNorm(128) - ReLU - MaxPool(2, 2)
   - Conv(128, 256, 3, 1, 1) - BatchNorm(256) - ReLU
   - Conv(256, 256, 3, 1, 1) - BatchNorm(256) - ReLU - MaxPool(2, 2)
   - Conv(256, 512, 3, 1, 1) - BatchNorm(512) - ReLU
   - Conv(512, 512, 3, 1, 1) - BatchNorm(512) - ReLU - MaxPool(2, 2)
   - Conv(512, 512, 3, 1, 1) - BatchNorm(512) - ReLU
   - Conv(512, 512, 3, 1, 1) - BatchNorm(512) - ReLU - MaxPool(2, 2)
   - FC(0512, 4096) - ReLU - Dropout(0.5)
   - FC(4096, 4096) - ReLU - Dropout(0.5)
   - FC(4096, 10)
   ```

   You should use the cross-entropy loss `torch.nn.CrossEntropyLoss` at the end.

   [This experiment will take up to 1 hour on a CPU, so please be cautious of your time. If this running time is not bearable, you may cut the training set to 1/10, so only have ∼600 images per class instead of the regular ∼6000.]

2. (4 pts) Once you've done the above, the next goal is to inspect the training process. Create the following plots:

   (a) (1 pt) test accuracy vs the number of epochs (say 3 ∼ 5)

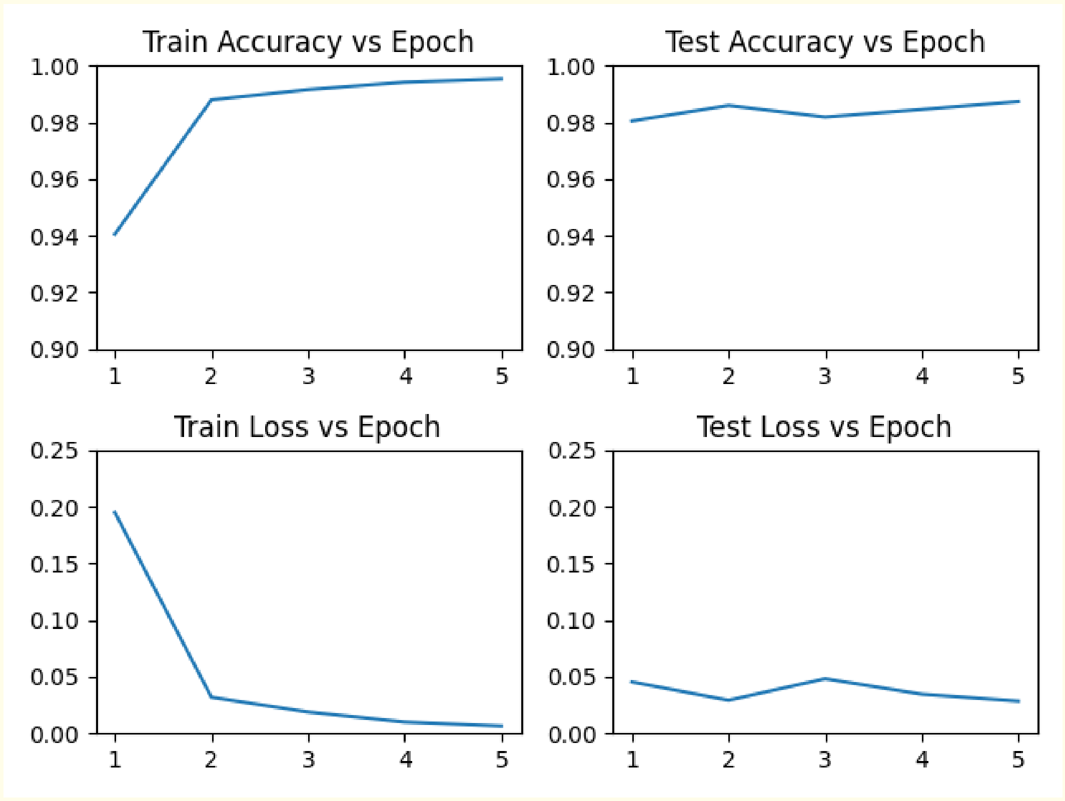
   (b) (1 pt) training accuracy vs the number of epochs

   (c) (1 pt) test loss vs the number of epochs

   (d) (1 pt) training loss vs the number of epochs

   [If running more than 1 epoch is computationally infeasible, simply run 1 epoch and try to record the accuracy/loss after every few minibatches.]

   Ans:



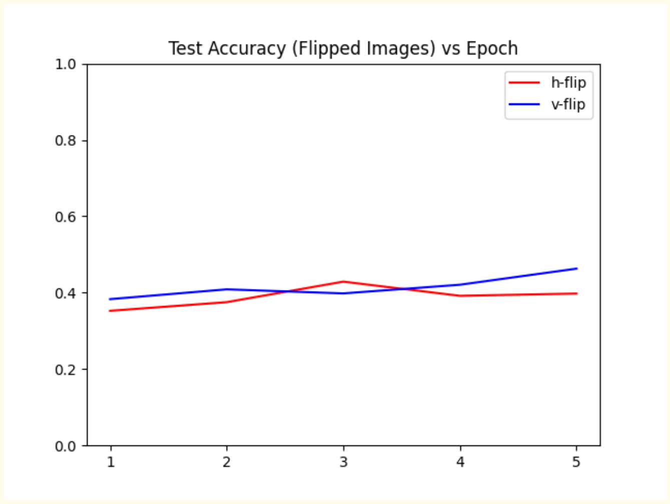
3. Then, it is time to inspect the generalization properties of your final model. Flip and blur the test set images using any python library of your choice, and complete the following:

   (e) (1 pt) test accuracy vs type of flip. Try the following two types of flipping: flip each image from left to right, and from top to bottom. Report the test accuracy after each flip. What is the effect?

   You can read this doc to learn how to build a complex transformation pipeline. We suggest the following command for performing flipping:

```
torchvision.transforms.RandomHorizontalFlip(p=1)
torchvision.transforms.RandomVerticalFlip(p=1)
```

   Ans: We can see that flipping our test images either horizontal or vertical reduces the accuracy to around 40 percent. Further, additional epochs of training don't seem to increase the accuracy.
   On deeper investigation, we can note that for horizontally-flipped images, our model predicts digits 0, 1, 4, and 8 with high accuracy and 5, 2, 7 with low accuracy (often mistaking one for the other). Similarly, for vertically-flipped images, our model predicts digits 3, 8, 1, 0, and 4 with high accuracy and 7, 2, 5, and 9 with low accuracy.
   This tracts given the horizontal and vertical symmetry of each digit.

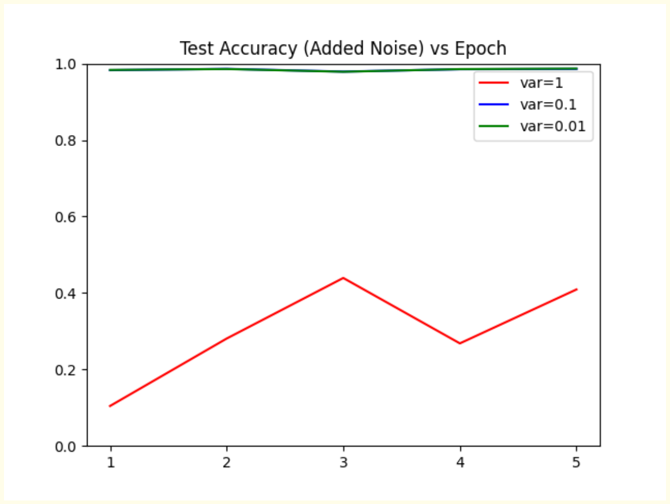


(f) (1 pt) test accuracy vs Gaussian noise. Try adding standard Gaussian noise to each test image with variance 0.01, 0.1, 1 and report the test accuracies. What is the effect?

For instance, you may apply a user-defined lambda as a new transform t which adds Gaussian noise with variance say 0.01:

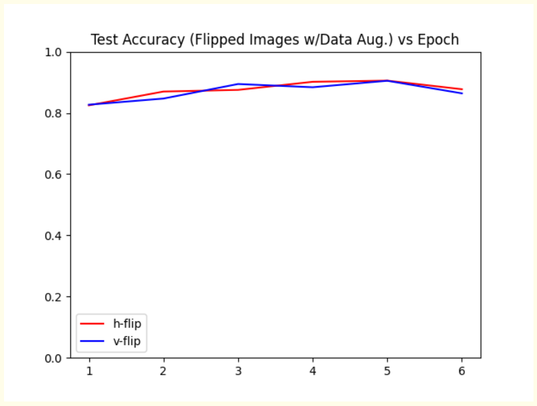
```
t = torchvision.transforms.Lambda(lambda x : x + 0.1*torch.randn_like(x))
```

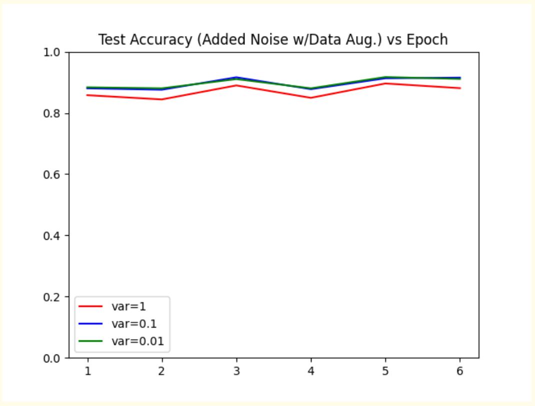
Ans: We can see that there is practically no perceptible difference in adding noise with variance 0.1 or 0.01. However, we see a significant dip in accuracy when we add noise with variance 1, implying that our model is not very resistant to added noise.



4. (2 pts) Lastly, let us verify the effect of regularization. Retrain your model with data augmentation and test again as in item 3 above (both e and f). Report the test accuracy and explain what kind of data augmentation you use in retraining.

Ans: First, we augment the data by adding noise (variance = 2) to all training images, then flip 30% of them horizontal or vertical respectively. Doing so, we see that our accuracy for random flips and noise improves significantly, even if training takes longer and is overall less accurate.
Note that we also augment our test set (no noise) with similar flippings.





Exercise 3: Regularization (4 pts)

Notation: For the vector $\mathbf{x}_i$, we use x_{ji} to denote its j-th element.

Overfitting to the training set is a big concern in machine learning. One simple remedy is through injecting noise: we randomly perturb each training data before feeding it into our machine learning algorithm. In this exercise you are going to prove that injecting noise to training data is essentially the same as adding some particular form of regularization. We use least-squares regression as an example, but the same idea extends to other models in machine learning almost effortlessly.

Recall that least-squares regression aims at solving:

$$\min_{\mathbf{w}\in\mathbb{R}^d} \sum_{i=1}^{n}(y_i - \mathbf{w}^\top \mathbf{x}_i)^2, \tag{3}$$

where $\mathbf{x}_i \in \mathbb{R}^d$ and $y_i \in \mathbb{R}$ are the training data. (For simplicity, we omit the bias term here.) Now, instead of using the given feature vector $\mathbf{x}_i$, we perturb it first by some independent noise $\boldsymbol{\epsilon}_i$ to get $\tilde{\mathbf{x}}_i = f(\mathbf{x}_i, \boldsymbol{\epsilon}_i)$, with different choices of the perturbation function f. Then, we solve the following **expected** least-squares regression problem:

$$\min_{\mathbf{w}\in\mathbb{R}^d} \sum_{i=1}^{n}\mathbf{E}[(y_i - \mathbf{w}^\top \tilde{\mathbf{x}}_i)^2], \tag{4}$$

where the expectation removes the randomness in $\tilde{\mathbf{x}}_i$ (due to the noise $\boldsymbol{\epsilon}_i$), and we treat $\mathbf{x}_i, y_i$ as fixed here. [To understand the expectation, think of n as so large that we have each data appearing repeatedly many times in our training set.]

1. (2 pts) Let $\tilde{\mathbf{x}}_i = f(\mathbf{x}_i, \boldsymbol{\epsilon}_i) = \mathbf{x}_i + \boldsymbol{\epsilon}_i$ where $\boldsymbol{\epsilon}_i \sim \mathcal{N}(\mathbf{0}, \lambda I)$ follows the standard Gaussian distribution. Simplify (4) as the usual least-squares regression (3), plus a familiar regularization function on $\mathbf{w}$.

   Ans: We say for a fixed i, there is a function $g(\epsilon) = y - \mathbf{w}^\top(\mathbf{x} + \epsilon)$, such that we are trying to find per i: $\mathbf{E}(g(\epsilon)^2)$. We note that $\mathbf{E}(g(\epsilon)^2) = \mathbf{E}(g(\epsilon))^2 + \mathbf{Var}(g(\epsilon))$ (by definition).
   Thus, we find:

$$\mathbf{E}(g(\epsilon)) = \mathbf{E}(y - \mathbf{w}^\top(\mathbf{x}+\epsilon)) = y - \mathbf{w}^\top\mathbf{E}(\mathbf{x}+\epsilon) = y - \mathbf{w}^\top(\mathbf{x}+\mathbf{E}(\epsilon)) = y - \mathbf{w}^\top\mathbf{x} \tag{5}$$

   by properties of expectation with $\mathbf{E}(\epsilon) = 0$.

$$\mathbf{Var}(g(\epsilon)) = \mathbf{Var}(y - \mathbf{w}^\top(\mathbf{x}+\epsilon)) = \mathbf{Var}(\mathbf{w}^\top(\mathbf{x}+\epsilon)) = \mathbf{w}\mathbf{w}^\top\mathbf{Var}(\mathbf{x}+\epsilon) = \mathbf{w}\mathbf{w}^\top\mathbf{Var}(\epsilon) = \lambda\mathbf{w}\mathbf{w}^\top \tag{6}$$

   by properties of variance with $\mathbf{Var}(\epsilon) = \lambda$.
   Putting this together:

$$\mathbf{E}(g(\epsilon)^2) = (y - \mathbf{w}^\top\mathbf{x})^2 + \lambda\mathbf{w}\mathbf{w}^\top \tag{7}$$

And we can write (4) as:

$$\min_{\mathbf{w}\in\mathbb{R}^d} \sum_{i=1}^{n}(y_i - \mathbf{w}^\top \mathbf{x}_i)^2 + \lambda||\mathbf{w}||_2^2 \tag{8}$$

And this is Tikhonov Regularization as seen in Lecture 2, Slide 17.

2. (2 pts) Let $\tilde{\mathbf{x}}_i = f(\mathbf{x}_i, \boldsymbol{\epsilon}_i) = \mathbf{x}_i \odot \boldsymbol{\epsilon}_i$, where $\odot$ denotes the element-wise product and $p\epsilon_{ji} \sim$ Bernoulli(p) independently for each j. That is, with probability $1-p$ we reset x_{ji} to 0 and with probability p we scale x_{ji} as x_{ji}/p. Note that for different training data $\mathbf{x}_i$, $\boldsymbol{\epsilon}_i$'s are independent. Simplify (4) as the usual least-squares regression (3), plus a different regularization function on $\mathbf{w}$ (that may also depend on $\mathbf{x}$). [This way of injecting noise, when applied to the weight vector $\mathbf{w}$ in a neural network, is known as Dropout (DropConnect).]

Ans: Similarly we say for a fixed i, there is a function $g(\epsilon) = y - \mathbf{w}^\top(\mathbf{x}\odot\epsilon)$, such that we are trying to find per i: $\mathbf{E}(g(\epsilon)^2)$. We note that $\mathbf{E}(g(\epsilon)^2) = \mathbf{E}(g(\epsilon))^2 + \mathbf{Var}(g(\epsilon))$ (by definition).
First, we can say $\mathbf{E}(\epsilon_j) = \mathbf{E}(p\epsilon_j/p) = 1$ and $\mathbf{Var}(\epsilon_j) = \mathbf{Var}(p\epsilon_j/p) = (1-p)/p$ using properties of Bernoulli. Thus, we find:

$$\mathbf{E}(g(\epsilon)) = \mathbf{E}(y - \mathbf{w}^\top(\mathbf{x}\odot\epsilon)) = y - \mathbf{w}^\top\mathbf{E}(\mathbf{x}\odot\epsilon)$$

For each j, $\mathbf{E}(\mathbf{x}_j\cdot\epsilon_j) = x_j$. And because for each j the value of $x_j\epsilon_j$ is independent, we can write:

$$\mathbf{E}(g(\epsilon)) = y - \mathbf{w}^\top\mathbf{x} \tag{9}$$

Similarly, we can say that $\mathbf{Var}(\mathbf{x}_j\cdot\epsilon_j) = \mathbf{x}_j^2(1-p)/p$, $\mathbf{Var}(\mathbf{x}\odot\epsilon) = \mathbf{x}\mathbf{x}^\top(1-p)/p$.

$$\mathbf{Var}(g(\epsilon)) = \mathbf{Var}(y - \mathbf{w}^\top(\mathbf{x}\odot\epsilon)) = \mathbf{w}\mathbf{w}^\top\mathbf{Var}(\mathbf{x}\odot\epsilon) = \mathbf{w}\mathbf{w}^\top\mathbf{x}\mathbf{x}^\top(1-p)/p \tag{10}$$

Putting this together:

$$\mathbf{E}(g(\epsilon)^2) = (y - \mathbf{w}^\top\mathbf{x})^2 + \mathbf{w}\mathbf{w}^\top\mathbf{x}\mathbf{x}^\top(1-p)/p \tag{11}$$

And we can write (4) as:

$$\min_{\mathbf{w}\in\mathbb{R}^d} \sum_{i=1}^{n}\left[(y_i - \mathbf{w}^\top\mathbf{x}_i)^2 + ||\mathbf{x}_i||_2^2||\mathbf{w}||_2^2(1-p)/p\right] \tag{12}$$