Homework 2 – Deep Learning (CS/DS 541, Murai, Fall 2023)

1. **Stochastic Gradient Descent** [15 points, on paper] First, consider the case where the mini-batch consists of a single observation $\widetilde{n} = 1$ sampled uniformly at random **with replacement**.

   (a) In this case, we will define an epoch as a sequence of n optimization iterations. What is the probability p_{never} that one observation is never sampled during one epoch? (Write it as a function of n). [4 points]. *Answer:* $p_{\text{never}} = (1 - 1/n)^n$

   (b) Using the fact that $\lim_{n\to\infty}(1 + 1/x)^x = e$, prove that, for large x, $p_{\text{never}} \approx 0.3679$. [3 points]. *Answer:* There are many ways to solve this question; we show a short proof.
   First, we prove an intermediate limit by replacing x by $\frac{1}{u}$. When $x \to \infty$, we have $u = 1/x \to 0$.

$$\lim_{x\to\infty} 1 + 1/x^x = \lim_{u\to 0} (1 + u)^{\frac{1}{u}} = e \tag{1}$$

$$\tag{2}$$

   Now we take the limit of p_{never} and substitute $u = -\frac{1}{n}$:

$$\lim_{n\to\infty} p_{\text{never}} = \lim_{n\to\infty} (1 + u)^{-\frac{1}{u}} = \left(\lim_{n\to\infty} (1 + u)^{\frac{1}{u}}\right)^{-1} = \frac{1}{e}. \tag{3}$$

Now, let's go back to the typical case where a minibatch of size $\widetilde{n} > 1$ is sampled without replacement. A number of recent empirical results show that the right amount of gradient variance in SGD is essential to train some types of deep neural networks. In particular, (Goyal et al., NeurIPS 2017) trains ResNet-50 on ImageNet in 1 hour by adjusting the learning rates as a function of minibatch size to keep the gradient noise constant. The authors argue: *"When the minibatch size is multiplied by k, multiply the learning rate by k."*

   (d) Let the variance of the gradient computed over one random observation be $\text{Var}(\nabla f(\mathbf{x}^{(i)}, y^i)) = \sigma^2$. What is the variance of the gradient computed over a minibatch? [2 points]. *Answer:* Let $\widetilde{n}$ be the size of the minibatch. The variance of the gradient computer over a minibatch of size $\widetilde{n}$ is

$$\text{Var}\Big(\frac{1}{\widetilde{n}} \sum_{i=1}^{\widetilde{n}} \nabla f(\hat{y}^{(i)}; y^i)\Big) = \frac{1}{\widetilde{n}^2}\widetilde{n}\sigma^2 = \frac{\sigma^2}{\widetilde{n}}$$

   .

   (e) Considering that the minibatch gradient is an average of i.i.d. random variables, what happens to the variance when you multiply the minibatch size by k? [4 points]. *Answer:* Accordingly, multiplying the batch size by k, the corresponding variance will be divided by k.

   (f) Why is the claim in the paper incorrect? How can you change the learning rate so that the gradient noise stays constant? [2 points]. *Answer:* Let ϵ be the original learning rate. When a minibatch of size $\widetilde{n}$ is used, the change in the parameter models is given by

$$\epsilon\Big(\frac{1}{\widetilde{n}} \sum_{i=1}^{\widetilde{n}} \nabla f(\hat{y}^{(i)}; y^i)\Big)\Big),$$

   and its variance is

$$\epsilon^2 \frac{\sigma^2}{\widetilde{n}}.$$

   When increasing the minibatch size and the learning rate k-fold, the variance of the SGD step becomes

$$(\epsilon k)^2 \frac{\sigma^2}{k\widetilde{n}} = k\epsilon^2 \frac{\sigma^2}{\widetilde{n}},$$

   hence the variance is not the same as in the original setting. To fix this, the learning rate should be multiplied by $\sqrt{k}$.

2. **XOR problem** [10 points, on paper]: Show (by deriving the gradient, setting to 0, and solving mathematically, not in Python) that the values for $\mathbf{w} = (w_1, w_2)$ and b that minimize the function $J(\mathbf{w}, b)$ in Equation 6.1 (in the *Deep Learning* textbook) are: $w_1 = 0$, $w_2 = 0$, and $b = 0.5$. *Answer*: Let $\mathbf{w} = [w_1, w_2]^\top$.

$$\begin{aligned} J(w_1, w_2, b) &= \frac{1}{4}\sum(\mathbf{x}^{(i)^\top}\mathbf{w} + b - y^{(i)})^2 \\ &= \frac{1}{4}\big((w_1 * 0 + w_2 * 0 + b - 0)^2 + (w_1 * 1 + w_2 * 0 + b - 1)^2 + \\ &\qquad (w_1 * 0 + w_2 * 1 + b - 1)^2 + (w_1 * 1 + w_2 * 1 + b - 0)^2\big) \end{aligned}$$

Now we find each partial derivative, set to 0, and solve:

$$\frac{\partial J}{\partial w_1} = \frac{1}{2}(w_1 + b - 1 + w_1 + w_2 + b) \tag{4}$$
$$= \frac{1}{2}(2w_1 + w_2 + 2b - 1) \tag{5}$$
$$\frac{\partial J}{\partial w_2} = \frac{1}{2}(w_2 + b - 1 + w_1 + w_2 + b) \tag{6}$$
$$= \frac{1}{2}(2w_2 + w_1 + 2b - 1) \tag{7}$$
$$\frac{\partial J}{\partial b} = \frac{1}{2}(b + w_1 + b - 1 + w_2 + b - 1 + w_1 + w_2 + b) \tag{8}$$
$$= \frac{1}{2}(2w_1 + 2w_2 + 4b - 2) \tag{9}$$

Setting to 0 and then subtracting Eq. 7 from 5, we have:

$$w_1 - w_2 = 0 \tag{10}$$
$$w_1 = w_2 \tag{11}$$

Substituting $w_2 = w_1$ into Eq. 9 and setting to 0, we obtain:

$$\frac{1}{2}(4w_1 + 4b - 2) = 2w_1 + 2b - 1 = 0 \tag{12}$$
$$w_1 = \frac{1 - 2b}{2} \tag{13}$$

Finally, we substitute back into Eq. 7 and set to 0:

$$\frac{1}{2}(3(1 - 2b)/2 + 2b - 1) = 0 \tag{14}$$
$$3/2 - 3b + 2b - 1 = 0 \tag{15}$$
$$b = 1/2 \tag{16}$$

and finally we deduce from Eq. 9:

$$2w_1 + 2(1/2) - 1 = 0 \tag{17}$$
$$w_1 = w_2 = 0 \tag{18}$$

3. L_2**-regularized Linear Regression via Stochastic Gradient Descent** [20 points, in Python]: Train a 2-layer neural network (i.e., linear regression) for age regression using the same data as in homework 1. Your prediction model should be $\hat{y} = \mathbf{x}^\top\mathbf{w} + b$. You should regularize $\mathbf{w}$ but not b. Note that, in contrast to Homework 1, this model includes a bias term.

   Instead of optimizing the weights of the network with the closed formula, use stochastic gradient descent (SGD). There are several different hyperparameters that you will need to choose:

- Mini-batch size $\tilde{n}$.
- Learning rate ϵ.
- Number of epochs.
- L_2 Regularization strength α.

In order not to cheat (in the machine learning sense) – and thus overestimate the performance of the network – it is crucial to optimize the hyperparameters **only** on a *validation set.* (The training set would also be acceptable but typically leads to worse performance.) To create a validation set, simply set aside a fraction (e.g., 20%) of the `age_regression_Xtr.npy` and `age_regression_ytr.npy` to be the validation set; the remainder (80%) of these data files will constitute the "actual" training data. While there are fancier strategies (e.g., Bayesian optimization – another probabilistic method, by the way!) that can be used for hyperparameter optimization, it's common to just use a grid search over a few values for each hyperparameter. In this problem, you are required to explore systematically (e.g., using nested `for` loops) at least 4 different parameters for each hyperparameter.

Performance evaluation: Once you have tuned the hyperparameters and optimized the weights so as to minimize the cost on the validation set, then: (1) **stop** training the network and (2) evaluate the network on the **test** set. Report the performance in terms of *unregularized* MSE.

Solution: Here are two of the key methods, where `alpha` is the regularization strength parameter:

```
def f_MSE_unreg (w, b, X, y, alpha):
    yhat = X.dot(w) + b
    return 0.5/len(y) * np.sum(np.power(yhat - y, 2))

def grad_f_MSE (w, b, X, y, alpha):
    yhat = X.dot(w) + b
    grad_w = 1./len(y) * np.dot(X, yhat - y) + alpha * w
    grad_b = 1./len(y) * np.sum(yhat - y)
    return grad_w, grad_b
```

4. **Linear Regression with MAE minimization via Gradient Descent** [10 points, on paper]: Consider the 2-layer neural network (i.e., linear regression) for age regression from the previous question. In contrast to the previous question:

   - this model does not include a regularization term.
   - we desire to train by optimizing the Mean Absolute Error (MAE), defined as $f_{\text{MAE}}(\mathbf{w}, b) = \frac{1}{n}\sum_{i=1}^{n} |(\mathbf{x}^{(i)})^\top \mathbf{w} + b - y^{(i)}|$.

   While the function $f(t) = |t|$ does not have a derivative at $t = 0$, in practice we can arbitrarily define $\nabla_t f(0) = 0$ to implement gradient descent (a prediction should rarely be exactly equal to the ground-truth value). Using that trick:

   (a) Show that the gradient $\nabla_b f_{\text{MAE}}(\mathbf{w}, b) = \frac{1}{n}(n_+ - n_-)$, where n_+ (respectively, n_-) is the number of cases where the prediction overestimates (respectively, underestimates) y. [3 pts] *Answer:* The gradient can be computed separately for each instance i and then aggregated.

   Note that $|t|$ is equal to t if $t > 0$ and equal to $-t$ if $t < 0$. Hence, if $(\mathbf{x}^{(i)})^\top \mathbf{w} + b - y^{(i)} > 0$ then

   $$\nabla_b |(\mathbf{x}^{(i)})^\top \mathbf{w} + b - y^{(i)}| = \nabla_b [(\mathbf{x}^{(i)})^\top \mathbf{w} + b - y^{(i)}] = 1.$$

   On the other hand, if $(\mathbf{x}^{(i)})^\top \mathbf{w} + b - y^{(i)} < 0$, then

   $$\nabla_b |(\mathbf{x}^{(i)})^\top \mathbf{w} + b - y^{(i)}| = \nabla_b [-((\mathbf{x}^{(i)})^\top \mathbf{w} + b - y^{(i)})] = -1.$$

   Summing over all instances yields $\nabla_b f_{\text{MAE}}(\mathbf{w}, b) = \frac{1}{n}(n_+ - n_-)$.

(b) Derive a similar expression for gradient $\nabla_{\mathbf{w}} f_{\text{MAE}}(\mathbf{w}, b)$. [3 pts] Using a similar idea, we obtain $\nabla_{\mathbf{w}} f_{\text{MAE}}(\mathbf{w}, b) = \frac{1}{n}(\sum_{\hat{y}^{(i)} > y^{(i)}} \mathbf{x}^{(i)} - \sum_{\hat{y}^{(i)} < y^{(i)}} \mathbf{x}^{(i)})$.

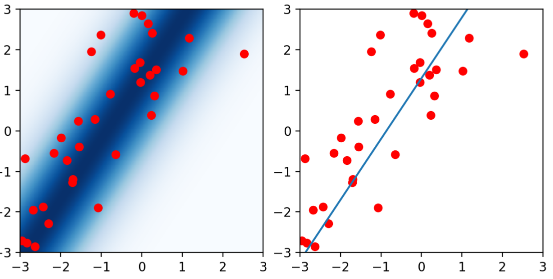
(c) Write a python function that takes as input: (i) the design matrix $\mathbf{X}$, (ii) the response vector $\mathbf{y}$, (iii) the current parameters $\mathbf{w}$ and b and, in turn, returns the updated parameter b after performing a gradient step. [4 pts] *Answer:*

```
def ComputeGradient( X, y, w, b):
    y_pred = X @ w + b
    error = y_pred - y
    is_pos_error = (error>0).squeeze()
    is_neg_error = (error<0).squeeze()

    return 1./len(y) * (np.sum(is_pos_error) - np.sum(is_neg_error))

def GradientDescentStep(X, y, w, b, learning_rate ):
    b_grad = ComputeGradient(X, y, w, b)
    return b - learning_rate * b_grad
```

5. **Linear-Gaussian prediction model** [14 points, on paper]:



Probabilistic prediction models enable us to estimate not just the "most likely" or "expected" value of the target y (see figure above, right), but rather an entire *probability distribution* about which target values are more likely than others, given input $\mathbf{x}$ (see figure above, left). In particular, a linear-Gaussian model is a Gaussian distribution whose expected value (mean μ) is a linear function of the input features $\mathbf{x}$, and whose variance is σ^2:

$$P(y \mid \mathbf{x}) = \mathcal{N}(\mu = \mathbf{x}^\top \mathbf{w}, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(y - \mathbf{x}^\top \mathbf{w})^2}{2\sigma^2}\right)$$

Note that, in general, σ^2 can also be a function of $\mathbf{x}$ (heteroscedastic case). Moreover, *non*-linear Gaussian models are also completely possible, e.g., the mean (and possibly the variance) of the Gaussian distribution is output by a deep neural network. However, in this problem, we will assume that μ is linear in $\mathbf{x}$, and that σ^2 is the same for all $\mathbf{x}$ (homoscedastic case).

MLE: The parameters of probabilistic models are commonly optimized by *maximum likelihood estimation* (MLE). (Another common approach is *maximum a posteriori* estimation, which allows the practitioner to incorporate a "prior belief" about the parameters' values.) Suppose the training dataset $\mathcal{D} = \{(\mathbf{x}^{(i)}, y^{(i)})\}_{i=1}^n$. Let the parameters/weights of the linear-Gaussian model be $\mathbf{w}$, such that the mean $\mu = \mathbf{x}^\top \mathbf{w}$. Prove that the MLE of $\mathbf{w}$ and σ^2 given $\mathcal{D}$ is:

$$\mathbf{w} = \left(\sum_{i=1}^n \mathbf{x}^{(i)} \mathbf{x}^{(i)\top}\right)^{-1} \left(\sum_{i=1}^n \mathbf{x}^{(i)} y^{(i)}\right)$$

$$\sigma^2 = \frac{1}{n} \sum_{i=1}^n (\mathbf{x}^{(i)\top} \mathbf{w} - y^{(i)})^2$$

Note that this solution – derived based on *maximizing* probability – is exactly the same as the optimal weights of a 2-layer neural network optimized to *minimize* MSE.

Hint: Follow the same strategy as the MLE derivation for a biased coin in Lecture 3. For a linear-Gaussian model, the argmax of the likelihood equals the argmax of the log-likelihood. The log of the Gaussian likelihood simplifies beautifully.

Solution:

$$\begin{aligned}
\log P(\mathcal{D} \mid \mathbf{w}, \sigma^2) &= \log \prod_{i=1}^{n} P(y^{(i)} \mid \mathbf{x}^{(i)}, \mathbf{w}, \sigma^2) \\
&= \sum_{i=1}^{n} \log P(y^{(i)} \mid \mathbf{x}^{(i)}, \mathbf{w}, \sigma^2) \\
&= \sum_{i=1}^{n} \log \frac{1}{\sqrt{2\pi\sigma^2}} \exp(-(\mathbf{x}^{(i)\top}\mathbf{w} - y^{(i)})^2/(2\sigma^2)) \\
&= \sum_{i=1}^{n} \left(-(\mathbf{x}^{(i)\top}\mathbf{w} - y^{(i)})^2/(2\sigma^2) - \log\sigma\right) + C \\
&= -\frac{1}{2\sigma^2}\sum_{i=1}^{n}(\mathbf{x}^{(i)\top}\mathbf{w} - y^{(i)})^2 \;\; - n\log\sigma + C
\end{aligned}$$

where C is some constant that does not depend on $\mathbf{w}$ or σ^2. To find the MLE, we differentiate the negative log-likelihood w.r.t. both $\mathbf{w}$ and σ, set to 0, and solve:

$$\begin{aligned}
\nabla_{\mathbf{w}} \log P(\mathcal{D} \mid \mathbf{w}, \sigma^2) &= \nabla_{\mathbf{w}} \left(\frac{1}{2\sigma^2}\sum_{i=1}^{n}(\mathbf{x}^{(i)\top}\mathbf{w} - y^{(i)})^2 \;\; + n\log\sigma + C\right) \\
0 &= \frac{1}{2\sigma^2}\sum_{i=1}^{n}\mathbf{x}^{(i)}(\mathbf{x}^{(i)\top}\mathbf{w} - y^{(i)}) \\
\mathbf{w} &= \left(\sum_{i=1}^{n}\mathbf{x}^{(i)}\mathbf{x}^{(i)\top}\right)^{-1}\left(\sum_{i=1}^{n}\mathbf{x}^{(i)}y^{(i)}\right) \\
\nabla_{\sigma} \log P(\mathcal{D} \mid \mathbf{w}, \sigma^2) &= \nabla_{\sigma} \left(\frac{1}{2\sigma^2}\sum_{i=1}^{n}(\mathbf{x}^{(i)\top}\mathbf{w} - y^{(i)})^2 \;\; + n\log\sigma + C\right) \\
0 &= -\frac{1}{\sigma^3}\sum_{i=1}^{n}(\mathbf{x}^{(i)\top}\mathbf{w} - y^{(i)})^2 + \frac{n}{\sigma} \\
\sigma^2 &= \frac{1}{n}\sum_{i=1}^{n}(\mathbf{x}^{(i)\top}\mathbf{w} - y^{(i)})^2
\end{aligned}$$