

# 2516 hw2

Zorina S

February 2023

## 1 Optimization

This week, we will continue investigating the properties of optimization algorithms, focusing on stochastic gradient descent and adaptive gradient descent methods.

We will continue using the linear regression model established in Homework 1. Given  $n$  pairs of input data with  $d$  features and scalar labels  $(\mathbf{x}_i, t_i) \in R^d \times R$ , we want to find a linear model  $f(\mathbf{x}) = \hat{\mathbf{w}}^T \mathbf{x}$  with  $\hat{\mathbf{w}} \in R^d$  such that the squared error on training data is minimized. Given a data matrix  $X \in R^{n \times d}$  and corresponding labels  $\mathbf{t} \in R^n$ , the objective function is defined as:

$$\mathcal{L} = \frac{1}{n} \|X\hat{\mathbf{w}} - \mathbf{t}\|_2^2 \quad (1)$$

### 1.1 Mini-Batch Stochastic Gradient Descent (SGD)

Mini-batch SGD performs optimization by taking the average gradient over a mini-batch, denoted  $\mathcal{B} \in R^{b \times d}$ , where  $1 < b \ll n$ . Each training example in the mini-batch, denoted  $\mathbf{x}_j \in \mathcal{B}$ , is randomly sampled without replacement from the data matrix  $X$ . Assume that  $X$  is full rank. Where  $\mathcal{L}$  denotes the loss on  $\mathbf{x}_j$ , the update for a single step of mini-batch SGD at time  $t$  with scalar learning rate  $\eta$  is:

$$\mathbf{w}_{t+1} \leftarrow \mathbf{w}_t - \frac{\eta}{b} \sum_{\mathbf{x}_j \in \mathcal{B}} \nabla_{\mathbf{w}_t} \mathcal{L}(\mathbf{x}_j, \mathbf{w}_t) \quad (2)$$

Mini-batch SGD iterates by randomly drawing mini-batches and updating model weights using the above equation until convergence is reached.

#### 1.1.1 Minimum Norm Solution blue[2pt]

Recall Question 3.3 from Homework 1. For an overparameterized linear model, gradient descent starting from zero initialization finds the unique minimum norm solution  $\mathbf{w}^*$  such that  $X\mathbf{w}^* = \mathbf{t}$ . Let  $\mathbf{w}_0 = \mathbf{0}$ ,  $d > n$ . Assume mini-batch SGD also converges to a solution  $\hat{\mathbf{w}}$  such that  $X\hat{\mathbf{w}} = \mathbf{t}$ . Show that mini-batch SGD solution is identical to the minimum norm solution  $\mathbf{w}^*$  obtained by gradient descent, i.e.,  $\hat{\mathbf{w}} = \mathbf{w}^*$ .

*Hint:* Be more specific as to what other solutions? Or is  $\mathbf{x}_j$  or  $\mathcal{B}$  contained in span of  $X$ ? Do the update steps of mini-batch SGD ever leave the span of  $X$ ?

**ANS:**

**output prompted by ChatGPT:**



Given  $n$  pairs of input data with  $d$  features and scalar labels  $(\mathbf{x}_i, t_i) \in \mathbb{R}^d \times \mathbb{R}$ , and a data matrix  $\mathbf{X} \in \mathbb{R}^{n \times d}$  and corresponding labels  $t \in \mathbb{R}^n$ , the objective function is defined as:

```
\begin{equation}
\mathcal{L} = \frac{1}{n} \|\mathbf{X} \hat{\mathbf{w}} - \mathbf{t}\|^2_2
\end{equation}. Each training example in the mini-batch, denoted  $\mathbf{x}_j \in \mathcal{B}$ , is randomly sampled without replacement from the data matrix  $\mathbf{X}$ . Assume that  $\mathbf{X}$  is full rank. Where  $\mathcal{L}$  denotes the loss on  $\mathbf{x}_j$ , the update for a single step of mini-batch SGD at time  $t$  with scalar learning rate  $\eta$  is:
\begin{equation}
\mathbf{w}_{t+1} \leftarrow \mathbf{w}_t - \frac{\eta}{n} \sum_{\mathbf{x}_j \in \mathcal{B}} (\mathbf{x}_j \mathbf{w}_t - t_j)
\end{equation}
Let  $\mathbf{w}_0 = \mathbf{0}$ . Assume mini-batch SGD also converges to a solution  $\hat{\mathbf{w}}$  such that  $\hat{\mathbf{w}} = \mathbf{t}$ . Show that mini-batch SGD solution is identical to the minimum norm solution  $\mathbf{w}^*$  obtained by gradient descent, i.e.,  $\mathbf{w}^* = \hat{\mathbf{w}}$ .
```



We want to show that the solution  $\hat{\mathbf{w}}$  obtained by mini-batch SGD is the same as the minimum norm solution  $\mathbf{w}^*$  obtained by gradient descent.



Let us first find the minimum norm solution using gradient descent. The objective function is:

```
\begin{equation}
\mathcal{L} = \frac{1}{2} \|\mathbf{w} - \mathbf{t}\|^2_2
\end{equation}
```

The gradient of this objective function with respect to  $\mathbf{w}$  is:

### Modified Output:

We want to show that the mini-batch SGD solution  $\hat{\mathbf{w}}$  is identical to the minimum norm solution  $\mathbf{w}^*$  obtained by gradient descent, i.e.,  $\hat{\mathbf{w}} = \mathbf{w}^*$ .

We can start by expressing  $\mathbf{w}^*$  in terms of  $\mathbf{X}$  and  $\mathbf{t}$  using the least squares solution formula:

$$\mathbf{w}^* = (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{t} \quad (3)$$

Since  $d > n$ ,  $\mathbf{X}^T \mathbf{X}$  is invertible and  $(\mathbf{X}^T \mathbf{X})^{-1}$  exists. Also note that  $\mathbf{X} \mathbf{w}^* = \mathbf{X}(\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{t} = \mathbf{t}$ .

Now, we can express the gradient of the loss with respect to  $\mathbf{w}$  as:

$$\nabla_{\mathbf{w}} \mathcal{L} = \frac{2}{n} \mathbf{X}^T (\mathbf{X} \mathbf{w} - \mathbf{t}) \quad (4)$$

Using this expression, we can write the update for mini-batch SGD as:

$$\mathbf{w}_{t+1} \leftarrow \mathbf{w}_t - \frac{\eta}{b} X^T (X\mathbf{w}_t - \mathbf{t}) \frac{b}{n} \quad (5)$$

Simplifying this expression, we get:

$$\mathbf{w}_{t+1} \leftarrow \mathbf{w}_t - \frac{\eta}{n} X^T (X\mathbf{w}_t - \mathbf{t}) \quad (6)$$

This is equivalent to the update rule for gradient descent, with a learning rate of  $\frac{\eta}{n}$ :

$$\mathbf{w}_{t+1} \leftarrow \mathbf{w}_t - \frac{\eta}{n} \nabla_{\mathbf{w}_t} \mathcal{L} \quad (7)$$

Therefore, the update rule for mini-batch SGD is identical to the update rule for gradient descent with a specific learning rate. This means that both algorithms will converge to the same solution.

Since both algorithms converge to the same solution, and we know that the solution obtained by gradient descent is the minimum norm solution, we can conclude that the solution obtained by mini-batch SGD is also the minimum norm solution. Therefore, we have shown that  $\hat{\mathbf{w}} = \mathbf{w}^*$ .

**My solution:**

$$\begin{aligned} 1.1.1. \quad \mathbf{w}_{t+1} &= \mathbf{w}_t - \frac{\eta}{b} \sum_{j \in B} \nabla_{\mathbf{w}_t} L(x_j, \mathbf{w}_t) \\ &= \mathbf{w}_t - \frac{\eta}{b} \sum_{j \in B} C \underbrace{\nabla_{\mathbf{w}_t} L(x_j, \mathbf{w}_t)}_{\in \mathbb{R}} \underbrace{\|x_j\|^2}_{\text{constant}} \\ &= \mathbf{w}_t - \frac{\eta}{b} \underbrace{\frac{2}{n} \sum_{j \in B} x_j}_{\in \mathbb{R}} \underbrace{(x_j^T \mathbf{w}_t - t_j)}_{\text{constant}} \end{aligned}$$

Here, each  $x_j \in B$  is randomly sampled without replacement from data matrix  $X$ , which means it is within the span of  $X$ , hence  $\nabla_{\mathbf{w}_t} L(x_j, \mathbf{w}_t)$  are linear combinations of vectors within  $X$ , the updated steps of minibatch does not leave the span of  $X$  ever.

As we assume minibatch SGD converges to a solution  $\tilde{\mathbf{w}}$  s.t.  $X\tilde{\mathbf{w}} = \mathbf{t}$ , we can also denote  $\tilde{\mathbf{w}} = X^T C$  where  $C \in \mathbb{R}$  and  $\tilde{\mathbf{w}} \in \text{span}(X)$ .

By hint, assume there is another solution  $\mathbf{w}$

$$\begin{aligned} (\tilde{\mathbf{w}} - \mathbf{w})^T \tilde{\mathbf{w}} &= C \tilde{\mathbf{w}} - \mathbf{w}^T X^T C = (X \tilde{\mathbf{w}} - X \mathbf{w})^T C = (C - \mathbf{t})^T C = 0 \\ \therefore (\tilde{\mathbf{w}} - \mathbf{w}) &\text{ are orthogonal to } \tilde{\mathbf{w}}, \text{ which means} \\ \| \mathbf{w} \|_2^2 &= \| (\mathbf{w} - \tilde{\mathbf{w}} + \tilde{\mathbf{w}}) \|_2^2 = \| \mathbf{w} - \tilde{\mathbf{w}} \|_2^2 + \| \tilde{\mathbf{w}} \|_2^2 \geq \| \tilde{\mathbf{w}} \|_2^2 \quad \text{for any } \mathbf{w}. \\ \therefore \tilde{\mathbf{w}} &\text{ is the minimum norm solution.} \end{aligned}$$

## 1.2 Adaptive Methods

We now consider the behavior of adaptive gradient descent methods. In particular, we will investigate the RMSProp method. Let  $w_i$  denote the  $i$ -th parameter. A scalar learning rate  $\eta$  is used. At time  $t$  for parameter  $i$ , the update step for RMSProp is shown by:

$$\begin{aligned} w_{i,t+1} &= w_{i,t} - \frac{\eta}{\sqrt{v_{i,t} + \epsilon}} \nabla_{w_{i,t}} \mathcal{L}(w_{i,t}) \\ v_{i,t} &= \beta(v_{i,t-1}) + (1 - \beta)(\nabla_{w_{i,t}} \mathcal{L}(w_{i,t}))^2 \end{aligned}$$

We begin the iteration at  $t = 0$ , and set  $v_{i,-1} = 0$ . The term  $\epsilon$  is a fixed small scalar used for numerical stability. The momentum parameter  $\beta$  is typically set such that  $\beta \geq 0.9$ . Intuitively, RMSProp adapts a separate learning rate in each dimension to efficiently move through badly formed curvatures (see lecture slides/notes).

### 1.2.1 Minimum Norm Solution blue[1pt]

Consider the overparameterized linear model ( $d > n$ ) for the loss function defined in Section 1. Assume the RMSProp optimizer converges to a solution. Provide a proof or counterexample for whether RMSProp always obtains the minimum norm solution.

*Hint:* Compute a simple 2D case. Let  $\mathbf{x}_1 = [2, 1]$ ,  $w_0 = [0, 0]$ ,  $t = [2]$ .

**ANS:**

**output prompted by ChatGPT:**

 consider the behavior of adaptive gradient descent methods. In particular, we will investigate the RMSProp method. Let  $w_i$  denote the  $i$ -th parameter. A scalar learning rate  $\eta$  is used. At time  $t$  for parameter  $i$ , the update step for RMSProp is shown by:

```
\begin{aligned} w_{i,t+1} &= w_{i,t} - \frac{\eta}{\sqrt{v_{i,t}} + \epsilon} \nabla_{w_i} \text{loss}(w_{i,t}) \\ v_{i,t+1} &= \beta v_{i,t} + (1-\beta) (\nabla_{w_i} \text{loss}(w_{i,t}))^2 \end{aligned}
```

We begin the iteration at  $t=0$ , and set  $v_{i,-1} = 0$ . The term  $\epsilon$  is a fixed small scalar used for numerical stability. The momentum parameter  $\beta$  is typically set such that  $\beta \geq 0.9$ . Intuitively, RMSProp adapts a separate learning rate in each dimension to efficiently move through badly formed curvatures.

Consider the overparameterized linear model ( $d>n$ ) for the loss function defined in Section [ref\(sec:opt\)](#). Assume the RMSProp optimizer converges to a solution. Provide a proof or counterexample for whether RMSProp always obtains the minimum norm solution.

Hint: Compute a simple 2D case. Let  $\mathbf{x}_1 = [2, 1]$ ,  $w_0 = [0, 0]$ ,  $t = 2$ .

 We can use the given update rule to compute the updates for the parameters  $w_i$  and  $v_i$  for the linear model defined in Section [ref\(sec:opt\)](#). However, we need to make some assumptions about the loss function and the dataset to proceed with the analysis. In particular, we assume that the dataset is well-behaved and that the loss function is convex with respect to the parameters  $w_j$ .

To simplify the analysis, let us consider a 2D case where  $d=2$  and  $n=1$ . We have one data point  $(\mathbf{x}_1, y_1)$  with  $\mathbf{x}_1 = [2, 1]$  and  $y_1 = 2$ . The loss function for this data point is given by

$$\text{loss}(w_1, w_2) = (w_1 x_{1,1} + w_2 x_{1,2} - y_1)^2 = (2w_1 + w_2 - 2)^2.$$

Note that the Hessian of this loss function is constant and positive definite, indicating that the loss function is convex.

Let us now apply the RMSProp update rule to this loss function. We initialize the parameters  $w_1$  and  $w_2$  to [Regenerate response](#) and rate  $\eta$  to  $0.1$ . We

The rest of the output are presented by the modified output(as my own solution) below.

### Modified Output:

We can use the given update rule to compute the updates for the parameters  $w_i$  and  $v_i$  for the linear model defined in Section 1. However, we need to make some assumptions about the loss function and the dataset to proceed with the analysis. In particular, we assume that the dataset is well-behaved and that the loss function is convex with respect to the parameters  $w_i$ .

To simplify the analysis, let us consider a 2D case where  $d = 2$  and  $n = 1$ . We have one data point  $(\mathbf{x}_1, y_1)$  with  $\mathbf{x}_1 = [2, 1]$  and  $y_1 = 2$ . The loss function for

$$\text{loss}(w_1, w_2) = (w_1 x_{1,1} + w_2 x_{1,2} - y_1)^2 = (2w_1 + w_2 - 2)^2.$$

this data point is given by

Note that the Hessian of this loss function is constant and positive definite, indicating that the loss function is convex.

Let us now apply the RMSProp update rule to this loss function. We initialize the parameters  $w_1$  and  $w_2$  to 0 and set the learning rate  $\eta$  to 0.1. We also set the momentum parameter  $\beta$  to 0.9 and the small scalar  $\epsilon$  to  $10^{-8}$ . Then, we compute the updates for  $w_1$  and  $w_2$  using the following equations:

$$v_{1,t} = 0.9v_{1,t-1} + 0.1(2x_{1,1}\mathcal{L}(w_{1,t}, w_{2,t}))^2 v_{2,t} = 0.9v_{2,t-1} + 0.1(2x_{1,2}\mathcal{L}(w_{1,t}, w_{2,t}))^2 w_{1,t+1} = w_{1,t} - \frac{0.1}{\sqrt{v_{1,t} + 10^{-8}}}(2x_{1,1}\mathcal{L}(w_{1,t}, w_{2,t})) w_{2,t+1} = w_{2,t} - \frac{0.1}{\sqrt{v_{2,t} + 10^{-8}}}(2x_{1,2}\mathcal{L}(w_{1,t}, w_{2,t}))$$

We can iteratively compute the updates for  $w_1$  and  $w_2$  until convergence. Note that since the loss function is convex and we have only one data point, the optimizer will converge to the minimum of the loss function.

However, it is not necessarily true that the RMSProp optimizer will always obtain the minimum norm solution. In the 2D case we considered, the minimum norm solution is given by  $w_1 = 1/3$  and  $w_2 = 4/3$ . It is possible that the RMSProp optimizer converges to a different solution with a larger norm, depending on the initialization and the learning rate schedule. Therefore, the answer to the question depends on the specific optimization problem and the behavior of the optimizer.

## 2 Gradient-based Hyper-parameter Optimization

In this problem, we will implement a simple toy example of *gradient-based hyper-parameter optimization*.

Often in practice, hyper-parameters are chosen by trial-and-error based on a model evaluation criterion. Instead, gradient-based hyper-parameter optimization computes gradient of the evaluation criterion w.r.t. the hyper-parameters and uses this gradient to directly optimize for the best set of hyper-parameters. For this problem, we will optimize for the learning rate of gradient descent in a regularized linear regression problem.

Specifically, given  $n$  pairs of input data with  $d$  features and scalar label  $(\mathbf{x}_i, t_i) \in R^d \times R$ , we wish to find a linear model  $f(\mathbf{x}) = \hat{\mathbf{w}}^\top \mathbf{x}$  with  $\hat{\mathbf{w}} \in R^d$  and a L2 penalty,  $\lambda \|\hat{\mathbf{w}}_2^2\|$ , that minimizes the squared error of prediction on the training samples.  $\lambda$  is a hyperparameter that modulates the impact of the L2 regularization on the loss function. Using the concise notation for the data matrix  $X \in R^{n \times d}$  and the corresponding label vector  $\mathbf{t} \in R^n$ , the squared error loss can be written as:

$$\tilde{\mathcal{L}} = \frac{1}{n} \|X\hat{\mathbf{w}} - \mathbf{t}\|_2^2 + \tilde{\lambda} \|\hat{\mathbf{w}}\|_2^2.$$

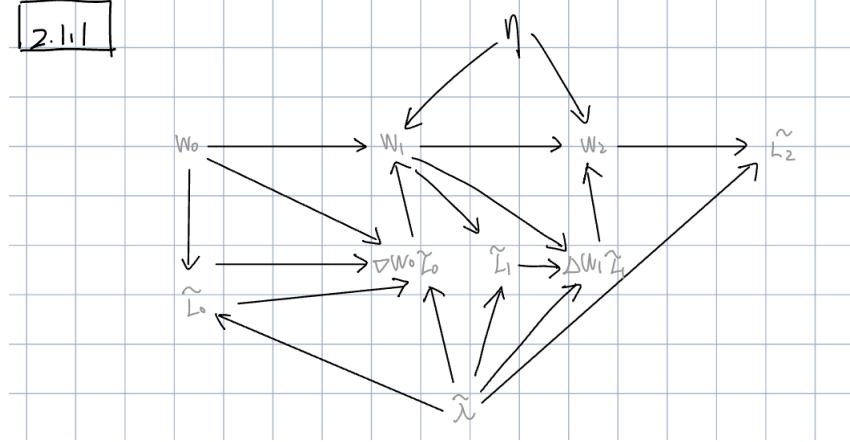
Starting with an initial weight parameters  $\mathbf{w}_0$ , gradient descent (GD) updates  $\mathbf{w}_0$  with a learning rate  $\eta$  for  $t$  number of iterations. Let's denote the weights after  $t$  iterations of GD as  $\mathbf{w}_t$ , the loss as  $\mathcal{L}_t$ , and its gradient as  $\nabla_{\mathbf{w}_t}$ . The goal is the find the optimal learning rate by following the gradient of  $\mathcal{L}_t$  w.r.t. the learning rate  $\eta$ .

## 2.1 Computation Graph

### 2.1.1 [0.5pt]

Consider a case of 2 GD iterations. Draw the computation graph to obtain the final loss  $\tilde{\mathcal{L}}_2$  in terms of  $\mathbf{w}_0, \nabla_{\mathbf{w}_0} \mathcal{L}_0, \mathcal{L}_0, \mathbf{w}_1, \mathcal{L}_1, \nabla_{\mathbf{w}_1} \mathcal{L}_1, \mathbf{w}_2, \lambda$  and  $\eta$ .

**My solution:**



### 2.1.2 blue[0.5pt]

Then, consider a case of  $t$  iterations of GD. What is the memory complexity for the forward-propagation in terms of  $t$ ? What is the memory complexity for using the standard back-propagation to compute the gradient w.r.t. the learning rate,  $\nabla_\eta \mathcal{L}_t$  in terms of  $t$ ?

*Hint:* Express your answer in the form of  $\mathcal{O}$  in terms of  $t$ .

**Output prompted by ChatGPT:**

given  $n$  pairs of input data with  $d$  features and scalar label  $\$y$ , we wish to find a linear model  $\hat{y}(\mathbf{x}) = \mathbf{w}^T \mathbf{x}$ . This can be done using the least squares method, which finds the line that minimizes the squared error of prediction on the training samples. The squared error loss function is given by:

$$\frac{1}{n} \sum_{i=1}^n (\hat{y}_i - y_i)^2 + \lambda \|\mathbf{w}\|_2^2$$

where  $\lambda$  is a hyperparameter that controls the strength of regularization. Using the concise notation for the data matrix  $\mathbf{X}$  and the corresponding label vector  $\mathbf{y}$ , the squared error loss can be written as:

$$\begin{aligned} & \begin{aligned}[t] & \text{\textbackslash begin\{align*\\} \\ & \text{\textbackslash Tilde\{loss\\} \\ & = \\ & \text{\textbackslash frac\{1\}\{n\}} \|\mathbf{X} \hat{\mathbf{w}} - \mathbf{y}\|_2^2 + \lambda \|\mathbf{w}\|_2^2. \\ & \text{\textbackslash end\{align*\\} \end{aligned} \end{aligned}$$

Starting with an initial weight parameters  $\mathbf{w}_0$ , gradient descent (GD) updates  $\mathbf{w}_0$  with a learning rate  $\eta$  for  $t$  number of iterations. Let's denote the weights after  $t$  iterations of GD as  $\mathbf{w}_t$ , the loss as  $\text{loss}_t$ , and its gradient as  $\nabla_{\mathbf{w}} \text{loss}_t$ . The goal is to find the optimal learning rate by following the gradient of  $\text{loss}_t$  w.r.t. the learning rate  $\eta$ .

\subsection{Computation Graph}

\subsubsection{\color{blue}[0.5pt] \LIV}

consider a case of  $t$  iterations of GD. What is the memory complexity for the forward-propagation in terms of  $t$ ? What is the memory complexity for using the standard back-propagation to compute the gradient w.r.t. the learning rate,  $\eta$  in terms of  $t$ ?

\textit{Hint}: Express your answer in the form of  $\mathcal{O}$  in terms of  $t$



The memory complexity for the forward propagation is  $\mathcal{O}(td)$ , since we need to store the weights  $\mathbf{w}_0, \mathbf{w}_1, \dots, \mathbf{w}_t$  at each iteration, and each weight vector has dimension  $d$ . Additionally, we need to store the intermediate computations involving the weights, such as  $\mathbf{w}_t^{\top} \mathbf{x}_i$  for each training example, which requires  $\mathcal{O}(tdn)$  memory.

The memory complexity for the standard backpropagation to compute  $\nabla_{\tilde{O}}(\text{loss}_t)$  is  $\mathcal{O}(td)$ , since we only need to store the gradients of the loss with respect to the weights,  $\nabla_{\mathbf{w}^0} \tilde{\text{loss}}_0$ ,  $\nabla_{\mathbf{w}^1} \tilde{\text{loss}}_1$ ,  $\dots$ ,  $\nabla_{\mathbf{w}_t} \tilde{\text{loss}}_t$ , and each gradient vector has dimension  $d$ . Therefore, the total memory required for backpropagation is  $\mathcal{O}(td^2)$ .

### **Modified Output:**

The memory complexity for the forward propagation is  $\mathcal{O}(td)$ , since we need to store the weights  $\mathbf{w}_0, \mathbf{w}_1, \dots, \mathbf{w}_t$  at each iteration, and each weight vector has dimension  $d$ . Additionally, we need to store the intermediate computations involving the weights, such as  $\mathbf{w}_t^\top \mathbf{x}_i$  for each training example, which requires  $\mathcal{O}(tdn)$  memory.

The memory complexity for the standard backpropagation to compute  $\nabla_\eta \mathcal{L}_t$  is  $\mathcal{O}(td)$ , since we only need to store the gradients of the loss with respect to the weights,  $\nabla_{\mathbf{w}0} \mathcal{L}_0, \nabla_{\mathbf{w}1} \mathcal{L}_1, \dots, \nabla_{\mathbf{w}t} \mathcal{L}_t$ , and each gradient vector has dimension  $d$ . Therefore, the total memory required for backpropagation is  $\mathcal{O}(td^2)$ .

The output works for me, so take it as my solution.

## 2.2 Optimal Learning Rates

In this section, we will take a closer look at the gradient w.r.t. the learning rate. To simplify the computation for this section, consider an unregularized loss function of the form  $\mathcal{L} = \frac{1}{n} \|X\hat{\mathbf{w}} - \mathbf{t}\|_2^2$ . Let's start with the case with only one GD iteration, where GD updates the model weights from  $\mathbf{w}_0$  to  $\mathbf{w}_1$ .

### 2.2.1 [1pt]

Write down the expression of  $\mathbf{w}_1$  in terms of  $\mathbf{w}_0$ ,  $\eta$ ,  $\mathbf{t}$  and  $X$ . Then use the expression to derive the loss  $\mathcal{L}_1$  in terms of  $\eta$ .

*Hint:* If the expression gets too messy, introduce a constant vector  $\mathbf{a} = X\mathbf{w}_0 - \mathbf{t}$   
**ANS:**

**output prompted by ChatGPT:**



consider an unregularized loss function of the form  $\text{loss} = \frac{1}{n} \|X\hat{\mathbf{w}} - \mathbf{t}\|_2^2$ . Let's start with the case with only one GD iteration, where GD updates the model weights from  $\mathbf{w}_0$  to  $\mathbf{w}_1$ .

```
\subsubsection{[1pt] L1}
Write down the expression of  $\mathbf{w}_1$  in terms of  $\mathbf{w}_0$ ,  $\eta$ ,  $\mathbf{t}$  and  $X$ . Then use the expression to derive the loss  $\text{loss}_1$  in terms of  $\eta$ .
\it Hint: If the expression gets too messy, introduce a constant vector  $\mathbf{a} = X\mathbf{w}_0 - \mathbf{t}$ 
```

< 2 / 2 > 

The update rule for one iteration of GD is:

```
\begin{aligned}
\mathbf{w}_1 &= \mathbf{w}_0 - \eta \nabla_{\mathbf{w}} \text{loss}(\mathbf{w}_0) \\
&= \mathbf{w}_0 - \eta \nabla_{\mathbf{w}} \text{loss}(\mathbf{w}_0) \frac{1}{n} \sum_{i=1}^n (\mathbf{w}_0 - \mathbf{t})_i^2 \\
&= \mathbf{w}_0 - \eta \frac{2}{n} \sum_{i=1}^n (\mathbf{w}_0 - \mathbf{t})_i \mathbf{w}_0^T
\end{aligned}
```

To derive the loss  $\text{loss}_1$  in terms of  $\eta$ , we substitute  $\mathbf{w}_1$  into the expression of  $\text{loss}$ :

```
\begin{aligned}
\text{loss}_1 &= \frac{1}{n} \sum_{i=1}^n (\mathbf{w}_1 - \mathbf{t})_i^2 \\
&= \frac{1}{n} \sum_{i=1}^n (\mathbf{w}_0 - \eta \nabla_{\mathbf{w}} \text{loss}(\mathbf{w}_0) \frac{1}{n} \sum_{j=1}^n (\mathbf{w}_0 - \mathbf{t})_j)_i^2 \\
&= \frac{1}{n} \sum_{i=1}^n (\mathbf{w}_0 - \mathbf{t} - \eta \frac{2}{n} \sum_{j=1}^n (\mathbf{w}_0 - \mathbf{t})_j)_i^2 \\
&= \frac{1}{n} \sum_{i=1}^n \left| \mathbf{w}_0 - \mathbf{t} - \eta \frac{2}{n} \sum_{j=1}^n (\mathbf{w}_0 - \mathbf{t})_j \right|^2
\end{aligned}
```

### Modified output:

The update rule for one iteration of GD is:

$$\mathbf{w}_1 = \mathbf{w}_0 - \eta \nabla_{\mathbf{w}} \mathcal{L}(\mathbf{w}_0) = \mathbf{w}_0 - \eta \nabla_{\mathbf{w}} \frac{1}{n} \sum_{i=1}^n \|\mathbf{w}_0 - \mathbf{t}\|_2^2 = \mathbf{w}_0 - \eta \frac{2}{n} X^T (X \mathbf{w}_0 - \mathbf{t})$$

To derive the loss  $\mathcal{L}_1$  in terms of  $\eta$ , we substitute  $\mathbf{w}_1$  into the expression of  $\mathcal{L}$ :

$$\begin{aligned}
\mathcal{L}_1 &= \frac{1}{n} \sum_{i=1}^n \|\mathbf{w}_1 - \mathbf{t}\|_2^2 \\
&= \frac{1}{n} \sum_{i=1}^n \|X(\mathbf{w}_0 - \eta \frac{2}{n} X^T (X \mathbf{w}_0 - \mathbf{t})) - \mathbf{t}\|_2^2 \\
&= \frac{1}{n} \sum_{i=1}^n \|X \mathbf{w}_0 - \mathbf{t} - \eta \frac{2}{n} X X^T (X \mathbf{w}_0 - \mathbf{t})\|_2^2 \\
&= \frac{1}{n} \sum_{i=1}^n \|\mathbf{a} - \eta \frac{2}{n} X X^T \mathbf{a}\|_2^2 \\
&= \frac{1}{n} (\mathbf{a} - \eta \frac{2}{n} X X^T \mathbf{a})^T (\mathbf{a} - \eta \frac{2}{n} X X^T \mathbf{a}) \\
&= \frac{1}{n} (\mathbf{a}^T \mathbf{a} - 2\eta \frac{2}{n} \mathbf{a}^T X X^T \mathbf{a} + \eta^2 \frac{4}{n^2} \mathbf{a}^T X X^T X X^T \mathbf{a})
\end{aligned}$$

The generated output fails to use the hint perfectly, as there is no substitution by  $\mathbf{a}$  in the equation of  $\mathbf{w}_1$ , then the value plugged into the  $\mathcal{L}_1$  is messy.

### My solution:

**2.2.1** With  $L = \frac{1}{n} \|X\hat{\omega} - t\|_2^2$ , we have:

$$\begin{aligned} W_1 &= W_0 - \eta \nabla_{W_0} L_0 = W_0 - \frac{2}{n} \eta X^T (XW_0 - t) \\ &= W_0 - \frac{2}{n} \eta X^T a \quad \text{by hint} \end{aligned}$$

$$\begin{aligned} \Rightarrow L_1 &= \frac{1}{n} \|XW_1 - t\|_2^2 = \frac{1}{n} \|X(W_0 - \frac{2}{n} \eta X^T a) - t\|_2^2 \\ &= \frac{1}{n} \left\| -\frac{2}{n} \eta X X^T a + XW_0 - t \right\|_2^2 \\ &= \frac{1}{n} \left\| -\frac{2}{n} \eta X X^T a + a \right\|_2^2 \\ &= \frac{1}{n} \left\| \left( -\frac{2}{n} \eta X X^T + I \right) a \right\|_2^2, \text{ by taking } a \text{ out} \\ &= \frac{1}{n} (a^T \left( -\frac{2}{n} \eta X X^T + I \right) \left( -\frac{2}{n} \eta X X^T + I \right)^T a) \\ &= \frac{1}{n} a^T \left( -\frac{2}{n} \eta X X^T + I \right)^2 a \end{aligned}$$

### 2.2.2 [0pt]

Determine if  $\mathcal{L}_1$  is convex w.r.t. the learning rate  $\eta$ .

*Hint:* A function is *convex* if its second order derivative is positive

pass

### 2.2.3 [1pt]

Write down the derivative of  $\mathcal{L}_1$  w.r.t.  $\eta$  and use it to find the optimal learning rate  $\eta^*$  that minimizes the loss after one GD iteration. Show your work.

**ANS:**

output prompted by ChatGPT:



Write down the derivative of  $\text{loss}_1$  w.r.t.  $\eta$  and use it to find the optimal learning rate  $\eta^*$  that minimizes the loss after one GD iteration. Show your work.



To find the derivative of  $\text{loss}_1$  w.r.t.  $\eta$ , we first expand the expression and simplify:

```
\begin{aligned}
\text{loss}_1 &= \frac{1}{n} (\mathbf{a}^T \mathbf{a} - 2 \eta \frac{2}{n} \mathbf{a}^T \mathbf{X} \mathbf{X}^T \\
&\quad \mathbf{a} + \eta^2 \frac{4}{n^2} \mathbf{a}^T \mathbf{X} \mathbf{X}^T \mathbf{X} \mathbf{X}^T \mathbf{a}) \\
&= \frac{1}{n} (\mathbf{a}^T \mathbf{a} - \frac{2}{n} \eta \mathbf{a}^T \mathbf{X} \mathbf{X}^T \mathbf{a} + \\
&\quad \mathbf{a}^T \mathbf{a} + \frac{4}{n^3} \eta^2 (\mathbf{a}^T \mathbf{X} \mathbf{X}^T \mathbf{a})^2
\end{aligned}
```

Taking the derivative w.r.t.  $\eta$ , we get:

```
\begin{aligned}
\frac{\partial \text{loss}_1}{\partial \eta} &= -\frac{2}{n} \mathbf{a}^T \mathbf{X} \mathbf{X}^T \mathbf{a} + \\
&\quad \frac{8}{n^3} \eta \mathbf{a}^T \mathbf{X} \mathbf{X}^T \mathbf{a}^2
\end{aligned}
```

Setting this derivative to zero and solving for  $\eta$ , we get:

```
\begin{aligned}
-\frac{2}{n} \mathbf{a}^T \mathbf{X} \mathbf{X}^T \mathbf{a} + \frac{8}{n^3} \eta \mathbf{a}^T \mathbf{X} \mathbf{X}^T \mathbf{a}^2 &= 0 \\
\eta \mathbf{a}^T \mathbf{X} \mathbf{X}^T \mathbf{a}^2 &= \frac{n}{4} (\mathbf{a}^T \mathbf{X} \mathbf{X}^T \mathbf{a}) \\
\eta &= \frac{n}{4} (\mathbf{a}^T \mathbf{X} \mathbf{X}^T \mathbf{a})
\end{aligned}
```

Substituting  $\mathbf{a} = \mathbf{X} \mathbf{w}_0 - \mathbf{t}$ , we have:

```
\begin{aligned}
\eta^* &= \frac{n}{4} ((\mathbf{X} \mathbf{w}_0 - \mathbf{t})^T \mathbf{X} \mathbf{X}^T (\mathbf{X} \mathbf{w}_0 - \mathbf{t})) \\
&= \frac{n}{4} ((\mathbf{X} \mathbf{w}_0)^T \mathbf{X} \mathbf{X}^T \mathbf{X} \mathbf{w}_0 - (\mathbf{X} \mathbf{w}_0)^T \mathbf{X} \mathbf{X}^T \mathbf{t} - \mathbf{t}^T \mathbf{X} \mathbf{X}^T \mathbf{X} \mathbf{w}_0 + \mathbf{t}^T \mathbf{X} \mathbf{X}^T \mathbf{t})
\end{aligned}
```

Similar problem with above, the hint is not used well, making the result messy.

**My solution:**

$$\begin{aligned}
 2.2.3 \quad \Delta_\eta L_1 &= \frac{1}{n} \times 2 \left( -\frac{2}{n} \eta X X^T a + a \right) \times \left( -\frac{2}{n} X X^T a \right) \\
 &= -\frac{4}{n^2} \left( -\frac{2}{n} \eta X X^T a + a \right) X X^T a \\
 &= -\frac{4}{n^2} \left( -\frac{2}{n} \eta X X^T + I \right) a X X^T a
 \end{aligned}$$

Use it to find the optimal learning rate  $\eta^*$ :

$$\text{set } \Delta_\eta L_1 = 0$$

$$\Rightarrow -\frac{2}{n} \eta X X^T a X X^T a = -a X X^T a$$

$$-\frac{2}{n} \eta = \frac{a X X^T a}{a (X X^T)^2 a}$$

$$\eta = \frac{n}{2} \frac{a X X^T a}{a (X X^T)^2 a} = \frac{n}{2} \frac{\|X^T a\|_2^2}{\|X X^T a\|_2^2}$$

## 2.3 Weight decay and L2 regularization

Although well studied in statistics, L2 regularization is usually replaced with explicit weight decay in modern neural network architectures:

$$w_{i+1} = (1 - \lambda) w_i - \eta \nabla \mathcal{L}_i(X)$$

In this question you will compare regularized regression of the form  $\mathcal{L} = \frac{1}{n} \|X \hat{w} - t\|_2^2 + \lambda \|\hat{w}\|_2^2$  with unregularized loss,  $\mathcal{L} = \frac{1}{n} \|X \hat{w} - t\|_2^2$ , accompanied by weight decay (equation 2.3).

### 2.3.1 blue[0.5pt]

Write down two expressions for  $w_1$  in terms of  $w_0$ ,  $\eta$ ,  $t$ ,  $\lambda$ ,  $\lambda$ , and  $X$ . The first one using  $\mathcal{L}$ , the second with  $\mathcal{L}$  and weight decay.

ANS:

$$2.3.1 \quad w_1 = w_0 - \frac{2}{n} \eta X^T (X w_0 - t) - 2 \eta \lambda w_0$$

$$w_1 = (1 - \lambda) w_0 - \frac{2}{n} \eta X^T (X w_0 - t)$$

### 2.3.2 blue[0.5pt]

How can you express  $\lambda$  (corresponding to L2 loss) so that it is equivalent to  $\lambda$  (corresponding to weight decay)?

*Hint:* Think about how you can express  $\lambda$  in terms of  $\lambda$  and another hyperparameter.

**ANS:**

2.3.2

By the above two equations, let them be connected by " $\hat{=}$ ".

$$W_0 - \eta \left[ \frac{2}{n} X^T (XW_0 - t) + 2\tilde{\lambda} W_0 \right] = (1-\lambda)W_0 - \frac{2}{n}\eta X^T (XW_0 - t)$$

$$W_0 - 2\eta X^T W_0 = (1-\lambda)W_0$$

$$-W_0 + 2\eta X^T W_0 = W_0 - W_0\lambda$$

$$\tilde{\lambda} = \frac{-\lambda W_0}{2\eta W_0}$$

$$= -\frac{\lambda}{2}\eta$$

### 2.3.3 [0pt]

Adaptive gradient update methods like RMSprop (equation 1.2) modulate the learning rate for each weight individually. Can you describe how L2 regularization is different from weight decay when adaptive gradient methods are used? In practice it has been shown that for adaptive gradients methods weight decay is more successful than l2 regularization.

pass

## 3 Convolutional Neural Networks

The last set of questions aims to build basic familiarity with convolutional neural networks (CNNs).

### 3.1 Convolutional Filters blue[0.5pt]

Given the input matrix  $I$  and filter  $J$  shown below, compute  $I * J$ , the output of the convolution operation (as defined in lecture 4). Assume zero padding is used such that the input and output are of the same dimension. What feature does this convolutional filter detect?

**ANS:**

$$\boxed{3.1} \quad I = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix} \quad J = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix} \quad I * J = \begin{bmatrix} 0 & 0 & -1 & -1 & -1 \\ -1 & -2 & 3 & 2 & 4 \\ 4 & 2 & 1 & 2 & -2 \\ -2 & 3 & 1 & 3 & -1 \\ 0 & -2 & 4 & -2 & 0 \end{bmatrix}$$

Detect Edges

### 3.2 Receptive Fields blue[0.5pt]

The receptive field of a neuron in a CNN is the area of the image input that can affect the neuron (i.e. the area a neuron can ‘see’). For example, a neuron in a  $3 \times 3$  convolutional layer is computed from an input area of  $3 \times 3$  of the input, so it’s receptive field is  $3 \times 3$ . However, as we go deeper into the CNN, the receptive field increases. One helpful resource to visualize receptive fields can be found at: <https://distill.pub/2019/computing-receptive-fields/>.

List 3 things that can affect the size of the receptive field of a neuron and briefly explain your answers.

**ANS prompted by ChatGPT:**



The receptive field of a neuron in a CNN is the area of the image input that can affect the neuron (i.e. the area a neuron can 'see'). For example, a neuron in a  $3 \times 3$  convolutional layer is computed from an input area of  $3 \times 3$  of the input, so it's receptive field is  $3 \times 3$ . However, as we go deeper into the CNN, the receptive field increases.

List 3 things that can affect the size of the receptive field of a neuron and briefly explain your answers.



1. Convolutional layer depth: As we move deeper into the CNN, the receptive field increases due to the stacking of multiple convolutional layers. Each layer in the CNN applies filters that capture increasingly complex features in the input image. These filters learn to detect patterns at different scales, and by stacking them, the receptive field of the neuron is increased to cover a larger area of the input image.
2. Kernel/Filter size: The size of the convolutional kernel used in the convolutional layers affects the size of the receptive field. Larger kernel sizes capture more spatial information, which can lead to a larger receptive field. For example, a  $5 \times 5$  kernel captures more spatial information than a  $3 \times 3$  kernel, so a neuron that uses a  $5 \times 5$  kernel has a larger receptive field than a neuron that uses a  $3 \times 3$  kernel.
3. Stride and pooling: The use of pooling layers and strides can also affect the receptive field. Pooling layers reduce the spatial dimension of the feature maps, which can lead to a smaller receptive field. However, strides can increase the receptive field by skipping over certain areas of the input image. By increasing the stride, the receptive field of the neuron is increased since it covers a larger area of the input image.



# Programming Part

## 4. Pooling and Upsampling

### 4.1 Output prompted by ChatGPT:



Using Python, Complete the model PoolUpsampleNet with 5 layer. Use the PyTorch layers nn.Conv2d, nn.ReLU, nn.BatchNorm2d, nn.Upsample, and nn.MaxPool2d. Your CNN should be configurable by parameters kernel, num\_in\_channels, num\_filters, and num\_colours. In the diagram, num\_in\_channels, num\_filters and num\_colours are denoted NIC, NF and NC respectively. Use the following parameterizations (if not specified, assume default parameters):

- nn.Conv2d: The number of input filters should match the second dimension of the input tensor (e.g. the first nn.Conv2d layer has NIC input filters). The number of output filters should match the second dimension of the output tensor (e.g. the first nn.Conv2d layer has NF output filters). Set kernel size to parameter kernel. Set padding to the padding variable included in the starter code.
- nn.BatchNorm2d: The number of features should match the second dimension of the output tensor (e.g. the first nn.BatchNorm2d layer has NF features).
- nn.Upsample: Use scaling factor = 2.
- nn.MaxPool2d: Use kernel size = 2.

Note: grouping layers according to the diagram (those not separated by white space) using the nn.Sequential containers will aid implementation of the forward method.

```
class PoolUpsampleNet(nn.Module): def __init__(self, kernel, num_filters,  
num_colours, num_in_channels):  
    super().__init__()
```

```
class PoolUpsampleNet(nn.Module):  
    def __init__(self, kernel, num_filters, num_colours, num_in_channels):  
        super().__init__()  
  
        # Useful parameters  
        padding = kernel // 2  
  
        # Define layers  
        self.conv1 = nn.Conv2d(num_in_channels, num_filters, kernel_size=kernel)  
        self.bn1 = nn.BatchNorm2d(num_filters)  
        self.relu1 = nn.ReLU()  
        self.pool1 = nn.MaxPool2d(kernel_size=2)  
  
        self.conv2 = nn.Conv2d(num_filters, num_filters, kernel_size=kernel)  
        self.bn2 = nn.BatchNorm2d(num_filters)  
        self.relu2 = nn.ReLU()  
        self.pool2 = nn.MaxPool2d(kernel_size=2)  
  
        self.conv3 = nn.Conv2d(num_filters, 2*num_filters, kernel_size=kernel)  
        self.bn3 = nn.BatchNorm2d(2*num_filters)  
        self.relu3 = nn.ReLU()  
        self.upsample1 = nn.Upsample(scale_factor=2, mode='nearest')  
  
        self.conv4 = nn.Conv2d(2*num_filters, num_filters, kernel_size=kernel)  
        self.bn4 = nn.BatchNorm2d(num_filters)  
        self.relu4 = nn.ReLU()  
        self.upsample2 = nn.Upsample(scale_factor=2, mode='nearest')  
  
        self.conv5 = nn.Conv2d(num_filters, num_colours, kernel_size=kernel)  
  
    def forward(self, x):  
        # Encoder  
        x = self.conv1(x)  
        x = self.bn1(x)  
        x = self.relu1(x)  
        x = self.pool1(x)  
  
        x = self.conv2(x)  
        x = self.bn2(x)  
        x = self.relu2(x)  
        x = self.pool2(x)  
  
        # Decoder  
        x = self.conv3(x)  
        x = self.bn3(x)  
        x = self.relu3(x)  
        x = self.upsample1(x)  
  
        x = self.conv4(x)  
        x = self.bn4(x)  
        x = self.relu4(x)  
        x = self.upsample2(x)  
  
        x = self.conv5(x)  
  
    return x
```

#### 4.1 My code:

```
class PoolUpsampleNet(nn.Module):
    def __init__(self, kernel, num_filters, num_colours, num_in_channels):
        super().__init__()

        # Useful parameters
        padding = kernel // 2

        ##### YOUR CODE GOES HERE #####
        self.layer1 = nn.Sequential(
            nn.Conv2d(in_channels=num_in_channels, out_channels=num_filters,
                      kernel_size=kernel, padding=padding),
            nn.MaxPool2d(kernel_size=2),
            nn.BatchNorm2d(num_features=num_filters),
            nn.ReLU(),
        )

        self.layer2 = nn.Sequential(
            nn.Conv2d(in_channels=num_filters, out_channels=num_filters*2,
                      kernel_size=kernel, padding=padding),
            nn.MaxPool2d(kernel_size=2),
            nn.BatchNorm2d(num_features=num_filters*2),
            nn.ReLU(),
        )

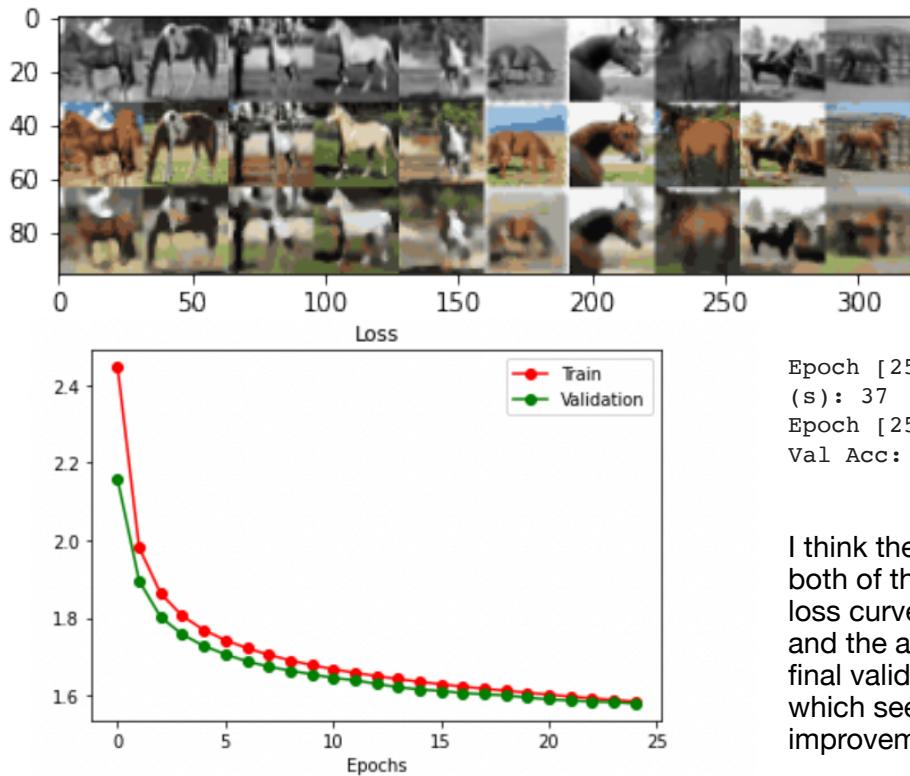
        self.layer3 = nn.Sequential(
            nn.Conv2d(in_channels=num_filters*2, out_channels=num_filters,
                      kernel_size=kernel, padding=padding),
            nn.Upsample(scale_factor=2),
            nn.BatchNorm2d(num_features=num_filters),
            nn.ReLU(),
        )

        self.layer4 = nn.Sequential(
            nn.Conv2d(in_channels=num_filters, out_channels=num_colours,
                      kernel_size=kernel, padding=padding),
            nn.Upsample(scale_factor=2),
            nn.BatchNorm2d(num_features=num_colours),
            nn.ReLU(),
        )

        self.layerFinal = nn.Conv2d(
            in_channels=num_colours, out_channels=num_colours,
            kernel_size=kernel, padding=padding)
        #####
    
    def forward(self, x):
        ##### YOUR CODE GOES HERE #####
        first = self.layer1(x)
        second = self.layer2(first)
        third = self.layer3(second)
        fourth = self.layer4(third)
        final = self.layerFinal(fourth)

        return final
#####
```

#### **4.2 generated images showing the trained result at the end**



Epoch [ 25/25 ], Loss: 1.5848, Time

(s): 37

Epoch [ 25/25 ], Val Loss: 1.5793,

Val Acc: 41.1%, Time(s): 38.29

I think the result are pretty good, as both of the training and validation loss curve show a decreasing trend and the accuracies increases. The final validation accuracy is 41.1% which seems good but need further improvement.

#### **4.3 Compute the number of weights, outputs, and connections in the model, as a function of NIC, NF and NC. Compute these values when each input dimension (width/height) is doubled. Report all 6 values.**

$$\begin{aligned}
 \text{Total # of weights} &= \text{NIC} \times \text{NF} \times \text{K}^2 + \text{NF} + 2\text{NF} + 2\text{NF}^2 \times \text{K}^2 + 2\text{NF} + 4\text{NF} \\
 &\quad + 2\text{NF}^2 \times \text{K}^2 + \text{NF} + 2\text{NF} + \text{NF} \times \text{NC} \times \text{K}^2 + \\
 &\quad \text{NC} + 2\text{NC} + \text{NC}^2 \times \text{K}^2 + \text{NC} \\
 &= \text{K}^2 \times [\text{NIC}(\text{NF}) + (\text{NF})(\text{NC}) + \text{NC}^2 + 4\text{NF}^2] + 12\text{NF} + 4\text{NC}
 \end{aligned}$$

$$\begin{aligned}
 \text{Total # of outputs} &= 32^2 \text{NF} + 16^2 \text{NF} + 16^2 \text{NF} + 16^2 \cdot 2\text{NF} + 8^2 \cdot 2\text{NF} + 8^2 \cdot 2\text{NF} \\
 &\quad + 8^2 \text{NF} + 16^2 \text{NF} + 16^2 \text{NF} + 16^2 \text{NC} + 32^2 \text{NC} + 32^2 \text{NC} \\
 &= 3328 \text{NC} + 2880 \text{NF}
 \end{aligned}$$

$$\begin{aligned}
 \text{Total # of connections} &= 32^2 \text{NIC} \times \text{NF} \times \text{K}^2 + 32^2 \text{NF} + 16^2 \text{NF} + 16^2 \text{K}^2 \text{NF} \cdot 2\text{NF} \\
 &\quad + 16^2 \cdot 2\text{NF} + 8^2 \cdot 2\text{NF} + 8^2 \cdot 2\text{NF} \cdot \text{NF} + 16^2 \text{NF} + 16^2 \text{NF} + \\
 &\quad 16^2 \text{K}^2 \text{NFNC} + 32^2 \text{NC} + 32^2 \text{K}^2 \times \text{NC}^2 \\
 &= \text{K}^2 (1024(\text{NIC})(\text{NF})) + 256(\text{NF})(\text{NC}) + 1024(\text{NC})^2 + 640(\text{NF})^2 \\
 &\quad + 2048 \text{NC} + 2432 \text{NF}
 \end{aligned}$$

If each input dimension is doubled,

Total # of weights remains unchanged.

Total # of outputs is 4x times of the previous number.

Total # of connections is 4x times of the previous number.

## 5. Strided and Transposed Dilated Convolutions

### 5.1 Output prompted by ChatGPT:



Using Python, Complete the model PoolUpsampleNet with 5 layer. Use the PyTorch layers nn.Conv2d, nn.ReLU, nn.BatchNorm2d, nn.Upsample, and nn.MaxPool2d. Your CNN should be configurable by parameters kernel, num\_in\_channels, num\_filters, and num\_colours. In the diagram, num\_in\_channels, num\_filters and num\_colours are denoted NIC, NF and NC respectively. Use the following parameterizations (if not specified, assume default parameters):

- nn.Conv2d: The number of input filters should match the second dimension of the input tensor (e.g. the first nn.Conv2d layer has NIC input filters). The number of output filters should match the second dimension of the output tensor (e.g. the first nn.Conv2d layer has NF output filters). Set kernel size to parameter kernel. Set padding to the padding variable included in the starter code.
- nn.BatchNorm2d: The number of features should match the second dimension of the output tensor (e.g. the first nn.BatchNorm2d layer has NF features).
- nn.Upsample: Use scaling factor = 2.
- nn.MaxPool2d: Use kernel size = 2.

Note: grouping layers according to the diagram (those not separated by white space) using the nn.Sequential containers will aid implementation of the forward method.

```
class PoolUpsampleNet(nn.Module):
    def __init__(self, kernel, num_filters, num_colours, num_in_channels):
        super().__init__()
```

```
class ConvTransposeNet(nn.Module):
    def __init__(self, kernel=3, num_filters=64, num_colours=3, num_in_channels=1):
        super().__init__()

        # Useful parameters
        stride = 2
        padding = kernel // 2
        output_padding = 1

        # Define layers
        self.conv1 = nn.Conv2d(num_in_channels, num_filters, kernel_size=kernel, stride=stride)
        self.bn1 = nn.BatchNorm2d(num_filters)
        self.relu1 = nn.ReLU()

        self.conv2 = nn.Conv2d(num_filters, num_filters*2, kernel_size=kernel, stride=stride)
        self.bn2 = nn.BatchNorm2d(num_filters*2)
        self.relu2 = nn.ReLU()

        self.conv3 = nn.ConvTranspose2d(num_filters*2, num_filters, kernel_size=kernel, stride=stride)
        self.bn3 = nn.BatchNorm2d(num_filters)
        self.relu3 = nn.ReLU()

        self.conv4 = nn.ConvTranspose2d(num_filters, num_colours, kernel_size=kernel, stride=stride)

    def forward(self, x):
        x = self.conv1(x)
        x = self.bn1(x)
        x = self.relu1(x)

        x = self.conv2(x)
        x = self.bn2(x)
        x = self.relu2(x)

        x = self.conv3(x)
        x = self.bn3(x)
        x = self.relu3(x)

        x = self.conv4(x)

    return x
```

 Regenerate response

## 5.1 My code:

```
class ConvTransposeNet(nn.Module):
    def __init__(self, kernel, num_filters, num_colours, num_in_channels):
        super().__init__()

        # Useful parameters
        stride = 2
        padding = kernel // 2
        output_padding = 1

        ##### YOUR CODE GOES HERE #####
        self.layer1 = nn.Sequential(
            nn.Conv2d(in_channels=num_in_channels, out_channels=num_filters,
                      kernel_size=kernel, stride=stride, padding=padding),
            nn.BatchNorm2d(num_features=num_filters),
            nn.ReLU(),
        )

        self.layer2 = nn.Sequential(
            nn.Conv2d(in_channels=num_filters, out_channels=num_filters*2,
                      kernel_size=kernel, stride=stride, padding=padding),
            nn.BatchNorm2d(num_features=num_filters*2),
            nn.ReLU(),
        )

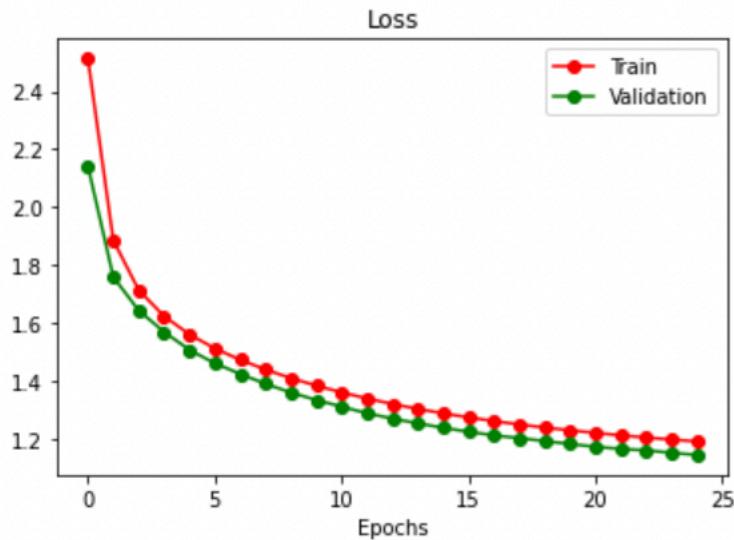
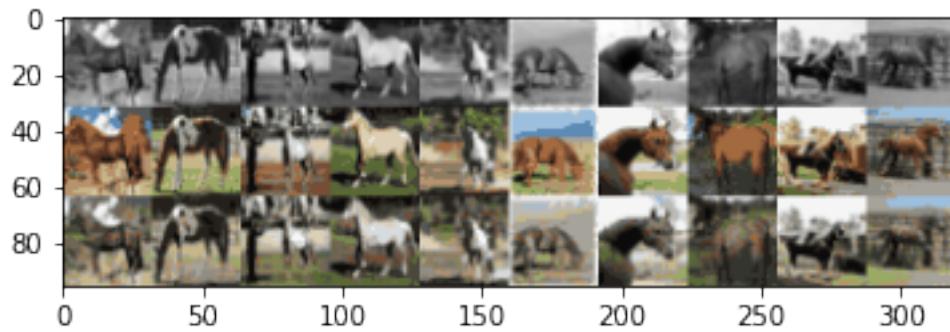
        self.layer3 = nn.Sequential(
            nn.ConvTranspose2d(in_channels=num_filters*2, out_channels=num_filters,
                      kernel_size=kernel, stride=stride, padding=padding,
                      dilation=1, output_padding=output_padding),
            nn.BatchNorm2d(num_features=num_filters),
            nn.ReLU(),
        )

        self.layer4 = nn.Sequential(
            nn.ConvTranspose2d(in_channels=num_filters, out_channels=num_colours,
                      kernel_size=kernel, stride=stride, dilation=1,
                      padding=padding, output_padding=output_padding),
            nn.BatchNorm2d(num_features=num_colours),
            nn.ReLU(),
        )

        self.layerFinal = nn.Conv2d(in_channels=num_colours, out_channels=num_colours,
                                  kernel_size=kernel, padding=padding)
        #####
    def forward(self, x):
        ##### YOUR CODE GOES HERE #####
        first = self.layer1(x)
        second = self.layer2(first)
        third = self.layer3(second)
        fourth = self.layer4(third)
        final = self.layerFinal(fourth)

        return final
#####
```

## 5.2 Output and training curve



### 5.3 How do the results compare to Section 4? Does the ConvTransposeNet model result in lower validation loss than the PoolUpsampleNet? Why may this be the case?

The training and validation loss resulted from ConvTransposeNet model are decreasing in a lower speed at the beginning, with the increase on number of epochs. However, as a result, the final validation loss here are lower than the result compare to Section 4, with a higher accuracy of 55.1% instead of 41.1% before.

The possible reason behind the situation is: upsample has nearest neighbour and bilinear modes according to *Upsampling in Core ML*. The height and width of the output would be reduced by max-pooling, which means there may be less parameter for the next layer and more features can be learnt using ConvTransposeNet model, which can lead to better result.

### 5.4 How would the padding parameter passed to the first two nn.Conv2d layers, and the padding and output padding parameters passed to the nn.ConvTranspose2d layers, need to be modified if we were to use a kernel size of 4 or 5.

If we were to use a kernel size of 4, and want to maintain the shapes of all tensors shown in Figure 1b, padding=1, output\_padding= 0.

If we were to use a kernel size of 5, padding=2 and output\_padding=1.

## 6 Skip Connections

### 6.1 output prompted by ChatGPT



Add a skip connection from the first layer to the last, second layer to the second last, etc. That is, the final convolution should have both the output of the previous layer and the initial greyscale input as input . Following the ConvTransposeNet class that you have completed, complete the init and forward methods of the UNet class. Hint: You will need to use the function torch.cat.



Here's the modified code for the `UNet` class with skip connections:



```
python
import torch.nn as nn

class UNet(nn.Module):
    def __init__(self, kernel=3, num_filters=64, num_colours=3, num_in_channels=1):
        super().__init__()

        # Useful parameters
        stride = 2
        padding = kernel // 2
        output_padding = 1

        # Define encoder layers
        self.conv1 = nn.Conv2d(num_in_channels, num_filters, kernel_size=kernel)
        self.bn1 = nn.BatchNorm2d(num_filters)
        self.relu1 = nn.ReLU()

        self.conv2 = nn.Conv2d(num_filters, num_filters*2, kernel_size=kernel)
        self.bn2 = nn.BatchNorm2d(num_filters*2)
        self.relu2 = nn.ReLU()

        self.conv3 = nn.Conv2d(num_filters*2, num_filters*4, kernel_size=kernel)
        self.bn3 = nn.BatchNorm2d(num_filters*4)
        self.relu3 = nn.ReLU()

        self.conv4 = nn.Conv2d(num_filters*4, num_filters*8, kernel_size=kernel)
        self.bn4 = nn.BatchNorm2d(num_filters*8)
        self.relu4 = nn.ReLU()

        self.conv5 = nn.Conv2d(num_filters*8, num_filters*8, kernel_size=kernel)
        self.bn5 = nn.BatchNorm2d(num_filters*8)
        self.relu5 = nn.ReLU()

        self.conv6 = nn.Conv2d(num_filters*8, num_filters*4, kernel_size=kernel)
        self.bn6 = nn.BatchNorm2d(num_filters*4)
        self.relu6 = nn.ReLU()

        self.conv7 = nn.Conv2d(num_filters*4, num_filters*2, kernel_size=kernel)
        self.bn7 = nn.BatchNorm2d(num_filters*2)
        self.relu7 = nn.ReLU()

        self.conv8 = nn.Conv2d(num_filters*2, num_filters, kernel_size=kernel)
        self.bn8 = nn.BatchNorm2d(num_filters)
        self.relu8 = nn.ReLU()

        self.conv9 = nn.Conv2d(num_filters, num_colours, kernel_size=kernel)
        self.bn9 = nn.BatchNorm2d(num_colours)
        self.relu9 = nn.ReLU()

    def forward(self, x):
        # Encoder path
        x1 = self.relu1(self.bn1(self.conv1(x)))
        x2 = self.relu2(self.bn2(self.conv2(x1)))
        x3 = self.relu3(self.bn3(self.conv3(x2)))
        x4 = self.relu4(self.bn4(self.conv4(x3)))
        x5 = self.relu5(self.bn5(self.conv5(x4)))
        x6 = self.relu6(self.bn6(self.conv6(x5)))
        x7 = self.relu7(self.bn7(self.conv7(x6)))
        x8 = self.relu8(self.bn8(self.conv8(x7)))
        x9 = self.relu9(self.bn9(self.conv9(x8)))

        # Decoder path with skip connections
        x = self.bn8(self.conv9(x9))
        x = torch.cat([x, x8], dim=1)
        x = self.relu8(self.bn8(self.conv8(x)))
        x = self.bn7(self.conv7(x))
        x = torch.cat([x, x7], dim=1)
        x = self.relu7(self.bn7(self.conv6(x)))
        x = self.bn6(self.conv6(x))
        x = torch.cat([x, x6], dim=1)
        x = self.relu6(self.bn6(self.conv5(x)))
        x = self.bn5(self.conv5(x))
        x = torch.cat([x, x5], dim=1)
        x = self.relu5(self.bn5(self.conv4(x)))
        x = self.bn4(self.conv4(x))
        x = torch.cat([x, x4], dim=1)
        x = self.relu4(self.bn4(self.conv3(x)))
        x = self.bn3(self.conv3(x))
        x = torch.cat([x, x3], dim=1)
        x = self.relu3(self.bn3(self.conv2(x)))
        x = self.bn2(self.conv2(x))
        x = torch.cat([x, x2], dim=1)
        x = self.relu2(self.bn2(self.conv1(x)))
        x = self.bn1(self.conv1(x))

        return x
```

Regenerate response

## 6.1 My code

```
class UNet(nn.Module):
    def __init__(self, kernel, num_filters, num_colours, num_in_channels):
        super().__init__()

        # Useful parameters
        stride = 2
        padding = kernel // 2
        output_padding = 1

        ##### YOUR CODE GOES HERE #####
        self.layer1 = nn.Sequential(
            nn.Conv2d(in_channels=num_in_channels, out_channels=num_filters,
                      kernel_size=kernel, stride=stride, padding=padding),
            nn.BatchNorm2d(num_features=num_filters),
            nn.ReLU(),
        )

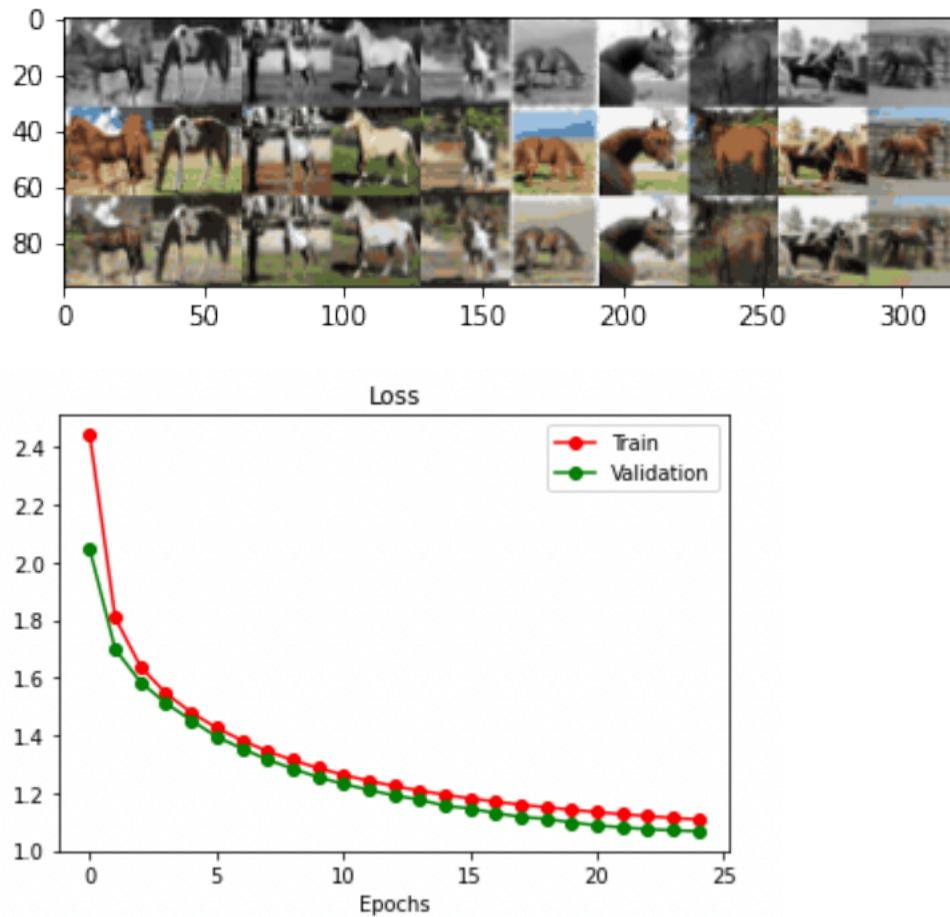
        self.layer2 = nn.Sequential(
            nn.Conv2d(in_channels=num_filters, out_channels=num_filters*2,
                      kernel_size=kernel, stride=stride, padding=padding),
            nn.BatchNorm2d(num_features=num_filters*2),
            nn.ReLU(),
        )

        self.layer3 = nn.Sequential(
            nn.ConvTranspose2d(in_channels=num_filters*2, out_channels=num_filters,
                      kernel_size=kernel, stride=stride, padding=padding,
                      output_padding=output_padding),
            nn.BatchNorm2d(num_features=num_filters),
            nn.ReLU(),
        )
        self.layer4 = nn.Sequential(
            nn.ConvTranspose2d(in_channels=num_filters*2, out_channels=num_colours,
                      kernel_size=kernel, stride=stride, padding=padding,
                      output_padding=output_padding),
            nn.BatchNorm2d(num_features=num_colours),
            nn.ReLU(),
        )
        self.layer5 = nn.Conv2d(in_channels=num_in_channels+num_colours,
                      out_channels=num_colours, kernel_size=kernel,
                      padding=padding)

        #####
    def forward(self, x):
        ##### YOUR CODE GOES HERE #####
        first = self.layer1(x)
        second = self.layer2(first)
        third = self.layer3(second)
        third = torch.cat([third, first], dim=1)
        fourth = self.layer4(third)
        fourth = torch.cat([fourth, x], dim=1)
        final = self.layer5(fourth)

        return final
#####
```

## 6.2 output and training curve



**6.3 How does the result compare to the previous model? Did skip connections improve the validation loss and accuracy? Did the skip connections improve the output qualitatively? How? Give at least two reasons why skip connections might improve the performance of our CNN models.**

The result here are better than the previous two, with the higher validation accuracy of 58.0% (instead of 55.1% in the previous part) and lower validation loss of 1.0672(instead of 1.1450%) , so skip connections improve the output qualitatively. The possible reason behind the situation can be attributed to

1. There are more parameters involved for training.
2. The gradient decent can perform better with the updated previous output layer during backward pass.
3. More info with the skip connections, like the info missed in the polling or convolution layers.