

CSC413A2

siweitang

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. For a refresher on optimization, refer to: <https://uoft-csc413.github.io/2023/assets/slides/lec03.pdf>.

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

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

a Mini-Batch Stochastic Gradient Descent (SGD)

Mini-batch SGD performs optimization by taking the average gradient over a mini-batch, denoted $\mathcal{B} \in \mathbb{R}^{b \times d}$, 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 η is:

$$\mathbf{w}_{t+1} \leftarrow \mathbf{w}_t - \frac{\eta}{b} \sum_{\mathbf{x}_j \in \mathcal{B}} \nabla_{\mathbf{w}_t} \mathcal{L}(\mathbf{x}_j, \mathbf{w}_t) = 0 \quad (2)$$

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

i 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 ?

Answer:

$$\begin{aligned} \nabla_{\mathbf{w}_t} \mathcal{L}(\mathbf{x}_j, \mathbf{w}_t) &= \nabla_{\mathbf{w}_t} \left(\frac{1}{n} \|\mathbf{w}_t^T \mathbf{x}_j - t_j\|_2^2 \right) \\ &= \frac{2}{n} \mathbf{x}_j (\mathbf{w}_t^T \mathbf{x}_j - t_j) \end{aligned}$$

Since $\mathbf{x}_j \in \mathcal{B}$, is randomly sampled without replacement from the data matrix X , so it is contained in span of X . Since $\nabla_{\mathbf{w}_t} \mathcal{L}(\mathbf{x}_j, \mathbf{w}_t)$ is the constant multiplier of \mathbf{x}_j and w_0 is 0, so the update steps of mini-batch SGD does not leave the span of X . If mini-batch SGD converges to other solution $\hat{\mathbf{w}}$, it must be also in the

span of X , in such a case, $\hat{\mathbf{w}} = X^T z$ for some $z \in \mathbb{R}^n$.
Assume there is other solution \mathbf{w} :

$$\begin{aligned}(\hat{\mathbf{w}} - \mathbf{w})^T \hat{\mathbf{w}} &= (\hat{\mathbf{w}} - \mathbf{w})^T X^T z \\&= (X^T \hat{\mathbf{w}} - X^T \mathbf{w})^T z \\&= (t - t)^T z \\&= 0\end{aligned}$$

That means $\hat{\mathbf{w}} - \mathbf{w}$ and $\hat{\mathbf{w}}$ are orthogonal.

$$\begin{aligned}\|\mathbf{w}\|^2 &= \|\mathbf{w} - \hat{\mathbf{w}} + \hat{\mathbf{w}}\|^2 \\&= \|\mathbf{w} - \hat{\mathbf{w}}\|^2 + \|\hat{\mathbf{w}}\|^2 \\&\geq \|\hat{\mathbf{w}}\|^2\end{aligned}$$

Therefore, the mini-batch SGD solution $\hat{\mathbf{w}}$ is the minimum norm solution, i.e., $\hat{\mathbf{w}} = \mathbf{w}^*$

chatGPT:

chatGPT has misundersstood the question, it may cause by I did not give the loss function in the prompt.

s

For an overparameterized linear model, gradient descent starting from zero initialization finds the unique minimum norm solution w^* such that $Xw^* = t$. Let $w_0 = 0$, $d > n$. Assume mini-batch SGD also converges to a solution \hat{w} such that $X\hat{w} = t$. Show that mini-batch SGD solution is identical to the minimum norm solution w^* obtained by gradient descent, i.e., $\hat{w} = w^*$



We start by noting that the optimization problem being solved by both gradient descent and mini-batch SGD is:



$$\text{minimize } \|Xw - t\|^2$$

where X is the data matrix of size $n \times d$, w is the weight vector of size $d \times 1$, and t is the target vector of size $n \times 1$.

Since the linear model is overparameterized (i.e., $d > n$), there are infinitely many solutions that can perfectly fit the training data. However, gradient descent with zero initialization finds the unique minimum norm solution w^* such that $Xw^* = t$. This means that w^* has the smallest possible norm among all solutions that perfectly fit the training data.

Now let's consider mini-batch SGD. At each iteration, mini-batch SGD updates the weight vector using a random subset (or mini-batch) of the training data. Let $w(k)$ be the weight vector after k iterations of mini-batch SGD, and let \hat{w} be the final weight vector obtained after convergence.

We can express the updates of mini-batch SGD as:

$$w(k+1) = w(k) - \alpha g(k)$$

where α is the learning rate, and $g(k)$ is the gradient of a randomly selected mini-batch at iteration k .

Since mini-batch SGD is a stochastic optimization algorithm, the weight vector at convergence \hat{w} may not be the same as the minimum norm solution w^* obtained by gradient descent. However, we will show that \hat{w} must have the same norm as w^* .

Let's consider the norm squared of the weight vector at iteration $k+1$:

$$\begin{aligned}\|w(k+1)\|^2 &= \|w(k) - \alpha g(k)\|^2 \\ &= \|w(k)\|^2 - 2\alpha w(k)^T g(k) + \alpha^2 \|g(k)\|^2\end{aligned}$$

Using the fact that $Xw(k) = t$ (since the weight vector at iteration k perfectly fits the training data), we can express the gradient as:

$$g(k) = 2X(k)^T(X(k)w(k) - t(k))$$

where $X(k)$ and $t(k)$ are the data and target matrices of the randomly selected mini-batch at iteration k .

Substituting this into the norm squared of the weight vector at iteration $k+1$, we get:

$$\begin{aligned}\|w(k+1)\|^2 &= \|w(k)\|^2 - 4\alpha(X(k)w(k) - t(k))^T X(k)^T w(k) + 4\alpha^2(X(k)w(k) - \\ &t(k))^T X(k)^T w(k) - t(k)\end{aligned}$$

Note that the second term in the above expression is zero, because $X(k)^T w(k) = X^T X w(k) = X^T t$, and $X(k)X(k)^T$ is the covariance matrix of the randomly selected mini-batch, which is proportional to the identity matrix and hence $X(k)X(k)^T w(k) = X(k)X^T t = X^T t$. Therefore, we have:

$$\|w(k+1)\|^2 = \|w(k)\|^2 + 4\alpha^2 \|X(k)w(k) - t(k)\|^2$$

Taking the expectation over the random mini-batches, we get:

$$E[\|w(k+1)\|^2] = \|w(k)\|^2 + 4\alpha^2 E[\|Xw(k) - t\|^2]$$

Note that $E[\|Xw(k) - t\|^2]$ is the mean squared error of the linear model at iteration k . Since mini-batch SGD converges to a

b 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 η is used. At time t for parameter i , the update step for RMSProp is shown by:

$$w_{i,t+1} = w_{i,t} - \frac{\eta}{\sqrt{v_{i,t}} + \epsilon} \nabla_{w_{i,t}} \mathcal{L}(w_{i,t}) \quad (3)$$

$$v_{i,t} = \beta(v_{i,t-1}) + (1 - \beta)(\nabla_{w_{i,t}} \mathcal{L}(w_{i,t}))^2 \quad (4)$$

We begin the iteration at $t = 0$, and set $v_{i,-1} = 0$. The term ϵ is a fixed small scalar used for numerical stability. The momentum parameter β 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).

i 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 $\mathbf{x}_1 = [2, 1]$, $w_0 = [0, 0]$, $t = [2]$.

Answer:

By question, $n = 1$, $d = 2$. Let $\beta = 0.9$, $\epsilon = 0.1$, $\eta = 0.1$

$$\begin{aligned}\nabla_{\mathbf{w}_0} \mathcal{L}(\mathbf{x}_1, \mathbf{w}_0) &= 2 \times \begin{pmatrix} 2 \\ 1 \end{pmatrix} \times -2 = \begin{pmatrix} -8 \\ -4 \end{pmatrix} \\ v_{0,0} &= 0.1 \times 64 = 6.4 \\ v_{1,0} &= 0.1 \times 16 = 1.6 \\ w_{0,1} &= 0 - \frac{0.1}{\sqrt{6.4 + 0.1}} \times (-8) = 0.3042 \\ w_{1,1} &= 0 - \frac{0.1}{\sqrt{1.6 + 0.1}} \times (-4) = 0.2931\end{aligned}$$

Since this solution is not in the span of $\left\{ \begin{pmatrix} 2 \\ 1 \end{pmatrix} \right\}$, so it is not the minimum solution. Thus, this counterexample violates solution.

chatGPT:

chatGPT has misundertsood the question, cannot answer the question.

s

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.



RMSProp is a popular optimization algorithm used for training neural networks, and it can also be used for optimizing overparameterized linear models. The algorithm uses a moving average of the squared gradients to adaptively adjust the learning rate for each weight during training. While RMSProp can help accelerate convergence and improve generalization, it does not guarantee convergence to the minimum norm solution for overparameterized linear models.



To see this, consider the overparameterized linear model:

$$Xw = t$$

where X is an $n \times d$ data matrix, w is a $d \times 1$ weight vector, and t is an $n \times 1$ target vector. Since $d > n$, there are infinitely many solutions that perfectly fit the training data.

The loss function for this model is:

$$L(w) = \|Xw - t\|^2$$

RMSProp updates the weight vector w at each iteration as:

$$w(k+1) = w(k) - \alpha / \sqrt{v(k) + \epsilon} * g(k)$$

where α is the learning rate, $v(k)$ is a moving average of the squared gradients, ϵ is a small constant to avoid division by zero, and $g(k)$ is the gradient of the loss function at iteration k .

Let w^* be the minimum norm solution that satisfies $Xw^* = t$. Note that w^* is unique due to the overparameterization of the model. We will show that RMSProp does not necessarily converge to w^* .

Assume that w^* has a non-zero component in the direction of a certain column of X , say the i -th column. That is, $w_i^* \neq 0$. Let $w(0) = 0$ be the initial weight vector. We can construct a sequence of training data $X(1), X(2), \dots$, such that each $X(k)$ differs from X by adding a small perturbation to the i -th column:

$$X(k) = X + \delta(k)e_i e_i^T$$

where $\delta(k)$ is a small positive scalar that decays to zero as k increases, e_i is the i -th standard basis vector, and $e_i e_i^T$ is the matrix with ones in the (i,i) position and zeros elsewhere.

It is clear that for any k , $X(k)w^* = t$, since adding a small perturbation to the i -th column of X does not affect the fact that $Xw^* = t$.

Now, let's consider the RMSProp updates for this sequence of training data. The gradient of the loss function with respect to the weight vector at iteration k is:

$$g(k) = 2X(k)^T(X(k)w(k) - t)$$

Substituting $X(k)w(k) = Xw(k)$ since the perturbation only affects the i -th column, we get:

$$g(k) = 2X(k)^T(Xw(k) - t)$$

Expanding this expression using the definition of $X(k)$, we have:

$$g(k) = 2(X^T X w(k) + \delta(k)e_i w(k)_i e_i^T - X^T t)$$

Using the fact that $Xw(k) = t$ for any k , we get:

$$g(k) = 2\delta(k)e_i w(k)_i e_i^T$$

The RMSProp update for the i -th weight at iteration k is:

$$w_i(k+1) = w_i(k) - \alpha / \sqrt{v_i(k) + \epsilon} * g_i(k)$$

where $v_i(k)$ is a moving average of the squared gradients for the i -th weight.

Since $g_i(k)$ only depends on the i -th weight and the i -th column of X , we can simplify the RMSProp update for the i -th weight

 Regenerate response

ii [0pt]

Consider the result from the previous section. Does this result hold true for other adaptive methods (Ada-grad, Adam) in general? Why might making learning rates independent per dimension be desirable?

2 Gradient-based Hyper-parameter Optimization

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

<https://uoft-csc413.github.io/2023/assets/slides/lec03.pdf>Lecture 3.

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

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

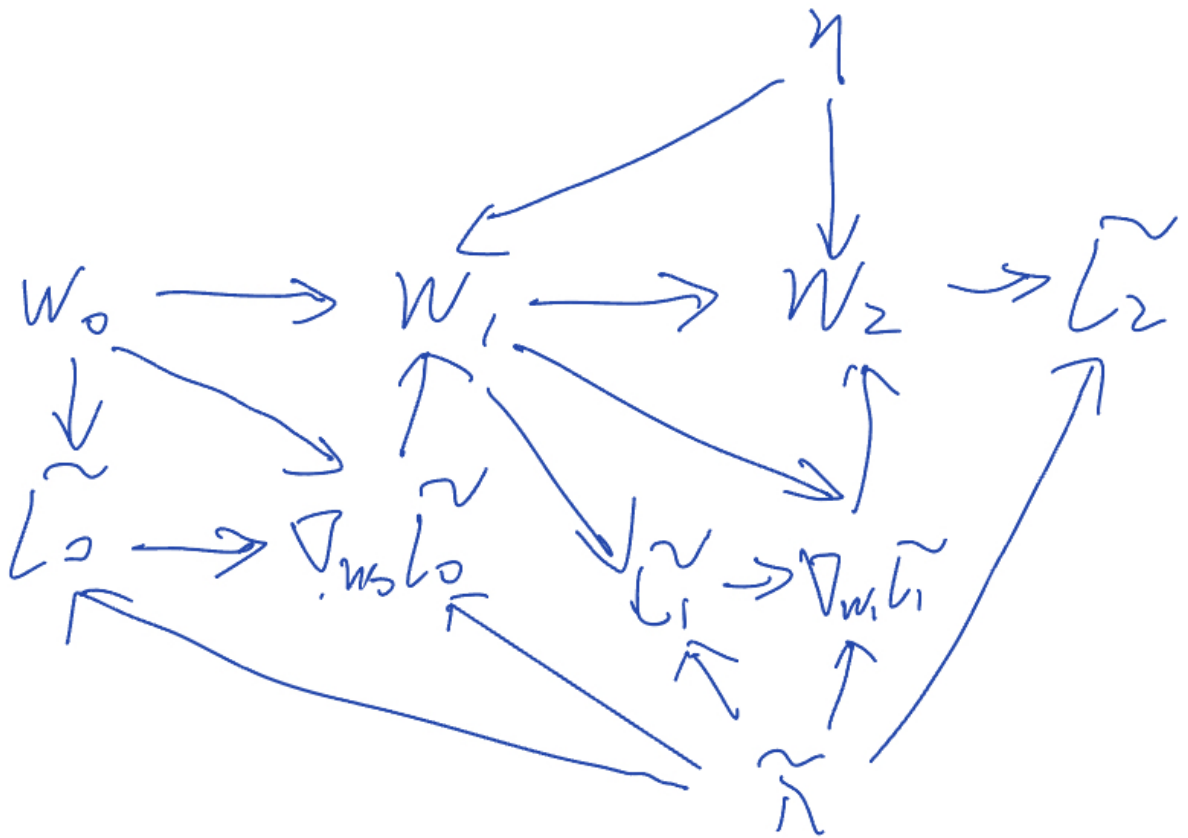
Starting with an initial weight parameters \mathbf{w}_0 , gradient descent (GD) updates \mathbf{w}_0 with a learning rate η for t number of iterations. Let's denote the weights after t iterations of GD as \mathbf{w}_t , the loss as \mathcal{L}_t , and its gradient as $\nabla_{\mathbf{w}_t}$. The goal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

a Computation Graph

i blue[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 $\mathbf{w}_0, \nabla_{\mathbf{w}_0} \tilde{\mathcal{L}}_0, \tilde{\mathcal{L}}_0, \mathbf{w}_1, \tilde{\mathcal{L}}_1, \nabla_{\mathbf{w}_1} \tilde{\mathcal{L}}_1, \mathbf{w}_2, \tilde{\lambda}$ and η .

Answer:



ii 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, $\nabla_{\eta} \tilde{\mathcal{L}}_t$ in terms of t ?

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

Answer:

For the forward-propagation the memory complexity is $\mathcal{O}(1)$. For the backward-propagation, it is $\mathcal{O}(t)$.

chatGPT:

It answers the question in correct way. Same as my answer.

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, $\nabla_{\eta} L \sim t$ in terms of t ?

Hint: Express your answer in the form of O in terms of t .

The memory complexity for the forward propagation in each iteration of gradient descent (GD) is $O(1)$. This is because the forward propagation involves computing the predicted output for a given input and weights, which can be done in a constant amount of memory.

The memory complexity for using the standard back-propagation to compute the gradient w.r.t. the learning rate, $\nabla_{\eta} L \sim t$, in each iteration of GD is $O(t)$. This is because the back-propagation involves computing the gradients of the loss with respect to the parameters, which requires storing the intermediate activations and gradients for each iteration. Thus, the memory required for back-propagation increases linearly with the number of iterations.

Therefore, the overall memory complexity for t iterations of GD using standard back-propagation is $O(t)$.

iii [0pt]

Explain one potential problem for applying gradient-based hyper-parameter optimization in more realistic examples where models often take many iterations to converge.

b 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 .

i blue[1pt]

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

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

Answer:

$$\begin{aligned}
w_1 &= w_0 - \frac{2\eta}{n} X^T (Xw_0 - t) = w_0 - \frac{2\eta}{n} X^T a \\
\mathcal{L}_1 &= \frac{1}{n} \|X\mathbf{w}_1 - \mathbf{t}\|_2^2 \\
&= \frac{1}{n} \|X(w_0 - \frac{2\eta}{n} X^T a) - \mathbf{t}\|_2^2 \\
&= \frac{1}{n} \left\| -\frac{2\eta}{n} X X^T a + Xw_0 - \mathbf{t} \right\|_2^2 \\
&= \frac{1}{n} \left\| -\frac{2\eta}{n} X X^T a + a \right\|_2^2 \\
&= \frac{1}{n} \left\| \left(I - \frac{2\eta}{n} X X^T \right) a \right\|_2^2 \\
&= \frac{1}{n} a^T \left(I - \frac{2\eta}{n} X X^T \right)^2 a
\end{aligned}$$

chatGPT:

chatGPT has misundersood the question, cannot answer the question.

S

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 $L = 1/n \|X\hat{w} - t\|^2$.

Let's start with the case with only one GD iteration, where GD updates the model weights from w_0 to w_1 .

Write down the expression of w_1 in terms of w_0 , η , t and X . Then use the expression to derive the loss L_1 in terms of η .

Hint: If the expression gets too messy, introduce a constant vector $a = Xw_0 - t$



Let's assume that the gradient descent algorithm takes one step, starting from an initial weight vector w_0 and using a learning rate of η . The update rule for the weight vector can be written as:



$$w_1 = w_0 - \eta \nabla L(w_0)$$

where $\nabla L(w_0)$ is the gradient of the loss function with respect to the weight vector evaluated at w_0 . For the unregularized loss function given as $L = 1/n \|X\hat{w} - t\|^2$, we have:

$$\nabla L(w_0) = 2/n X^T(Xw_0 - t)$$

Substituting this into the update rule, we get:

$$w_1 = w_0 - 2\eta/n X^T(Xw_0 - t)$$

Let's now define a constant vector $a = Xw_0 - t$. Substituting this into the expression above, we get:

$$w_1 = w_0 - 2\eta/n X^T a$$

We can now use the expression for w_1 to derive the loss L_1 in terms of η . Substituting w_1 into the unregularized loss function, we get:

$$L_1 = 1/n \|Xw_1 - t\|^2$$

Expanding this expression using the definition of w_1 and a , we get:

$$L_1 = 1/n \|X(w_0 - 2\eta/n X^T a) - t\|^2$$

Simplifying this expression, we get:

$$L_1 = 1/n \|Xw_0 - t - 2\eta/n X^T Xw_0 + 2\eta/n X^T t\|^2$$

Expanding the norm and using the fact that $\|a + b\|^2 = \|a\|^2 + 2a^T b + \|b\|^2$, we get:

$$L_1 = 1/n (\|Xw_0 - t\|^2 + 4\eta^2/n^2 \|X^T Xw_0\|^2 - 4\eta/n (X^T Xw_0)^T (Xw_0 - t) + \|X^T t\|^2)$$

Simplifying the middle term using the fact that $(A^T B)^T = B^T A$ and $(AB)^T = B^T A^T$, we get:

$$L_1 = 1/n (\|Xw_0 - t\|^2 + 4\eta^2/n^2 \|Xw_0\|^2 - 4\eta/n (X^T t)^T Xw_0 + \|X^T t\|^2)$$

Simplifying further using the fact that $X^T t$ is a constant vector, we get:

$$L_1 = 1/n (\|a\|^2 + 4\eta^2/n^2 \|Xw_0\|^2 - 4\eta/n (X^T t)^T Xw_0)$$

This is the expression for the loss L_1 in terms of η , which we obtained by computing the gradient of the unregularized loss function with respect to the learning rate.

ii [0pt]

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

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

iii blue[1pt]

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

Answer:

$$\begin{aligned}
\mathcal{L}_1 &= \frac{1}{n} a^T (I - \frac{2\eta}{n} X X^T)^2 a \\
\nabla_{\eta} \mathcal{L}_1 &= \frac{2}{n} a^T (I - \frac{2\eta}{n} X X^T) (-\frac{2}{n} X X^T) a = 0 \\
&= \frac{2}{n} a^T (-\frac{2}{n} X X^T + \frac{4\eta}{n^2} (X X^T)^2) a = 0 \\
\Rightarrow \eta &= \frac{n}{2} \frac{a^T X X^T a}{a^T (X X^T)^2 a} \\
&= \frac{n}{2} \frac{\|X^T a\|_2^2}{\|X X^T a\|_2^2}
\end{aligned}$$

chatGPT:

chatGPT has misundersstood the question, cannot answer the question.

s

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 $L = 1/n \|Xw - t\|^2$.

Let's start with the case with only one GD iteration, where GD updates the model weights from w_0 to w_1 . Write down the derivative of L w.r.t. η and use it to find the optimal learning rate η^* that minimizes the loss after one GD iteration. Show your work.



We have the unregularized loss function:



$$L(w) = 1/n * \|Xw - t\|^2$$

where w is the model weights, X is the input data, t is the target values, and n is the number of data points.

Let's consider one iteration of gradient descent, where the weights are updated from w_0 to w_1 as follows:

$$w_1 = w_0 - \eta * \nabla L(w_0)$$

where $\nabla L(w_0)$ is the gradient of the loss function with respect to the weights at w_0 , and η is the learning rate.

To find the derivative of L w.r.t. η , we can use the chain rule:

$$dL/d\eta = dL/dw_1 * dw_1/d\eta$$

where

$$\begin{aligned} dL/dw_1 &= d(1/n * \|Xw_1 - t\|^2)/dw_1 \\ &= 2/n * X^T(Xw_1 - t) \end{aligned}$$

and

$$dw_1/d\eta = - \nabla L(w_0)$$

🔄 Regenerate response

Therefore,

$$dL/d\eta = (2/n * X^T(Xw_1 - t)) * (-\nabla L(w_0))$$

Setting this derivative to zero to find the optimal learning rate:

$$0 = dL/d\eta = (2/n * X^T(Xw_1 - t)) * (-\nabla L(w_0))$$

This implies that either $\nabla L(w_0)$ or $X^T(Xw_1 - t)$ is zero. However, we want to avoid the trivial solution where the gradient is zero, which would mean we have reached a minimum.

Therefore, we can assume that $X^T(Xw_1 - t)$ is non-zero.

Solving for η , we get:

$$\eta^* = 2 / (n * \|X^T(Xw_1 - t)\|)$$

This is the optimal learning rate that minimizes the loss after one GD iteration.

Note that in practice, we would use multiple iterations of gradient descent, and the learning rate may need to be adjusted dynamically. Additionally, regularization terms may also be added to the loss function.

c Weight decay and L2 regularization

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

$$\mathbf{w}_{i+1} = (1 - \lambda)\mathbf{w}_i - \eta \nabla \mathcal{L}_i(X) \quad (5)$$

In this question you will compare regularized regression of the form $\tilde{\mathcal{L}} = \frac{1}{n} \|X\hat{\mathbf{w}} - \mathbf{t}\|_2^2 + \tilde{\lambda} \|\hat{\mathbf{w}}\|_2^2$ with unregularized loss, $\mathcal{L} = \frac{1}{n} \|X\hat{\mathbf{w}} - \mathbf{t}\|_2^2$, accompanied by weight decay (equation 5).

i blue[0.5pt]

Write down two expressions for \mathbf{w}_1 in terms of \mathbf{w}_0 , η , \mathbf{t} , λ , $\tilde{\lambda}$, and X . The first one using $\tilde{\mathcal{L}}$, the second with \mathcal{L} and weight decay.

Answer:

$$\begin{aligned}
\nabla_{w_0} \tilde{\mathcal{L}} &= \frac{2}{n} X^T (X \mathbf{w}_0 - \mathbf{t}) + 2\tilde{\lambda} \mathbf{w}_0 \\
\mathbf{w}_1 &= \mathbf{w}_0 - \eta \nabla_{w_0} \tilde{\mathcal{L}} \\
&= \mathbf{w}_0 - \frac{2\eta}{n} X^T (X \mathbf{w}_0 - \mathbf{t}) - 2\eta \tilde{\lambda} \mathbf{w}_0 \\
\nabla_{w_0} \mathcal{L}_0 &= \frac{2}{n} X^T (X \mathbf{w}_0 - \mathbf{t}) \\
\mathbf{w}_1 &= (1 - \lambda) \mathbf{w}_0 - \eta \nabla_{w_0} \mathcal{L}_0 \\
&= (1 - \lambda) \mathbf{w}_0 - \frac{2\eta}{n} X^T (X \mathbf{w}_0 - \mathbf{t})
\end{aligned}$$

ii blue[0.5pt]

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

Hint: Think about how you can express $\tilde{\lambda}$ in terms of λ and another hyperparameter.

Answer:

$$\begin{aligned}
\mathbf{w}_0 - 2\eta \tilde{\lambda} \mathbf{w}_0 &= (1 - \lambda) \mathbf{w}_0 \\
2\eta \tilde{\lambda} &= \lambda \\
\tilde{\lambda} &= \frac{\lambda}{2\eta}
\end{aligned}$$

iii [0pt]

Adaptive gradient update methods like RMSprop (equation 4) 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.

3 Convolutional Neural Networks

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

a Convolutional Filters blue[0.5pt]

Given the input matrix \mathbf{I} and filter \mathbf{J} shown below, compute $\mathbf{I} * \mathbf{J}$, 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?

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

Answer:

The filter does the horizontal edge detection.

b Size of Conv Nets blue[1pt]

CNNs provides several advantages over fully connected neural networks (FCNNs) when applied to image data. In particular, FCNNs do not scale well to high dimensional image data, which you will demonstrate below. Consider the following CNN architecture on the left: The input image has dimension 32×32 and is RGB (three channel). For ease of computation, assume all convolutional layers only have 1 output channel, and use 3×3 kernels. Assume zero padding is used in convolutional layers such that the output dimension is equal to the input dimension. Each max pooling layer has a filter size of 2×2 and a stride of 2. Furthermore, ignore all bias terms.

We consider an alternative architecture, shown on the right, which replaces convolutional layers with fully connected (FC) layers in an otherwise identical architecture. For both the CNN architecture and the FCNN architecture, compute the total number of neurons in the network, and the total number of trainable parameters. You should report four numbers in total. Finally, name one disadvantage of having more trainable parameters.

Answer:

For CNN

total number of neurons is $32 * 32 + 16 * 16 + 16 * 16 + 8 * 8 + 8 * 8 = 1664$

total number of trainable parameters is $(3 * 3 + 1) * 3 = 30$

For FCNN

total number of neurons is $32 * 32 + 16 * 16 + 16 * 16 + 8 * 8 + 8 * 8 = 1664$

total number of trainable parameters is $32 * 32 * 32 * 32 + 32 * 32 + 16 * 16 * 16 * 16 + 16 * 16 + 8 * 8 * 8 * 8 + 8 * 8 = 1119552$

More trainable parameters would increase the computation complexity.

c 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 3×3 convolutional layer is computed from an input area of 3×3 of the input, so it’s receptive field is 3×3 . 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.

Answer:

1. The stride value of max pooling layer
2. The size of filter/kernel
3. The number of layers

chatGPT:

chatGPT has misundertsood the question, cannot answer the question.



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 3×3 convolutional layer is computed from an input area of 3×3 of the input, so its receptive field is 3×3 . 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



1. Convolutional layer type: Different convolutional layer types have different ways of computing the output feature map, and this can affect the size of the receptive field. For example, a standard convolutional layer with a 3×3 filter will have a receptive field of 3×3 , while a dilated convolutional layer with a dilation rate of 2 and a 3×3 filter will have a receptive field of 7×7 .
2. Stride: The stride of the convolutional operation determines the amount of spatial downsampling in the output feature map. A larger stride will result in a smaller receptive field, since each neuron in the output feature map corresponds to a smaller area of the input. Conversely, a smaller stride will result in a larger receptive field, since each neuron in the output feature map corresponds to a larger area of the input.
3. Pooling: Pooling layers downsample the spatial dimensions of the feature maps by taking the maximum or average value in a neighborhood of neurons. This downsampling can increase the effective receptive field of neurons in the feature maps, since each neuron in the output feature map now corresponds to a larger area of the input. For example, a 2×2 max pooling layer applied to a 3×3 feature map will result in a 2×2 output feature map, where each neuron corresponds to a 2×2 area of the input. This means that the receptive field of each neuron in the output feature map is now 3×3 .



Image Colourization as Classification

In this section, we will perform image colourization using three convolutional neural networks (Figure c). Given a grayscale image, we wish to predict the color of each pixel. We have provided a subset of 24 output colours, selected using k-means clustering¹. The colourization task will be framed as a pixel-wise classification problem, where we will label each pixel with one of the 24 colours. For simplicity, we measure distance in RGB space. This is not ideal but reduces the software dependencies for this assignment.

We will use the CIFAR-10 data set, which consists of images of size 32×32 pixels. For most of the questions

¹https://en.wikipedia.org/wiki/K-means_clustering

we will use a subset of the dataset. The data loading script is included with the notebooks, and should download automatically the first time it is loaded.

Helper code for Section 4 is provided in , which will define the main training loop as well as utilities for data manipulation. Run the helper code to setup for this question and answer the following questions.

4 Pooling and Upsampling

a blue[0.5pt]

Complete the model `PoolUpsampleNet`, following the diagram in Figure ?? . Use the PyTorch layers `nn.Conv2d`, `nn.ReLU`, `nn.BatchNorm2d`², `nn.Upsample`³, 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.

Answer:

²<https://gauthamkumaran.com/batchnormalization/amp/>

³<https://machinethink.net/blog/coreml-upsampling/>

```

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.first = 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.second = 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.third = 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.forth = 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.last = 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.first(x)
        second = self.second(first)
        third = self.third(second)
        fourth = self.forth(third)
        return self.last(fourth)
        #####

```

chatGPT:

chatGPT can give the nearly accurate answer based on the information given in prompt.

s

Complete the model PoolUpsampleNet, following the diagram in Figure 1a. 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.



Here's the implementation of the PoolUpsampleNet model in PyTorch:

python

Copy code

```
import torch.nn as nn

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

        self.padding = (kernel - 1) // 2

        self.layer1 = nn.Sequential(
            nn.Conv2d(num_in_channels, num_filters, kernel_size=kernel, padding=self.padding),
            nn.BatchNorm2d(num_filters),
            nn.ReLU(),
```

Regenerate response

```

        nn.ReLU(),
        nn.MaxPool2d(kernel_size=2)
    )

    self.layer2 = nn.Sequential(
        nn.Conv2d(num_filters, num_filters, kernel_size=kernel, padding=self.pa
        nn.BatchNorm2d(num_filters),
        nn.ReLU(),
        nn.MaxPool2d(kernel_size=2)
    )

    self.layer3 = nn.Sequential(
        nn.Conv2d(num_filters, num_filters, kernel_size=kernel, padding=self.pa
        nn.BatchNorm2d(num_filters),
        nn.ReLU(),
        nn.Upsample(scale_factor=2, mode='nearest')
    )

    self.layer4 = nn.Sequential(
        nn.Conv2d(num_filters, num_filters, kernel_size=kernel, padding=self.pa
        nn.BatchNorm2d(num_filters),
        nn.ReLU(),
        nn.Upsample(scale_factor=2, mode='nearest')
    )

    self.layer5 = nn.Sequential(
        nn.Conv2d(num_filters, num_colors, kernel_size=kernel, padding=self.pad
        nn.BatchNorm2d(num_colors),
        nn.ReLU()
    )

def forward(self, x):
    x = self.layer1(x)
    x = self.layer2(x)
    x = self.layer3(x)
    x = self.layer4(x)
    x = self.layer5(x)
    return x

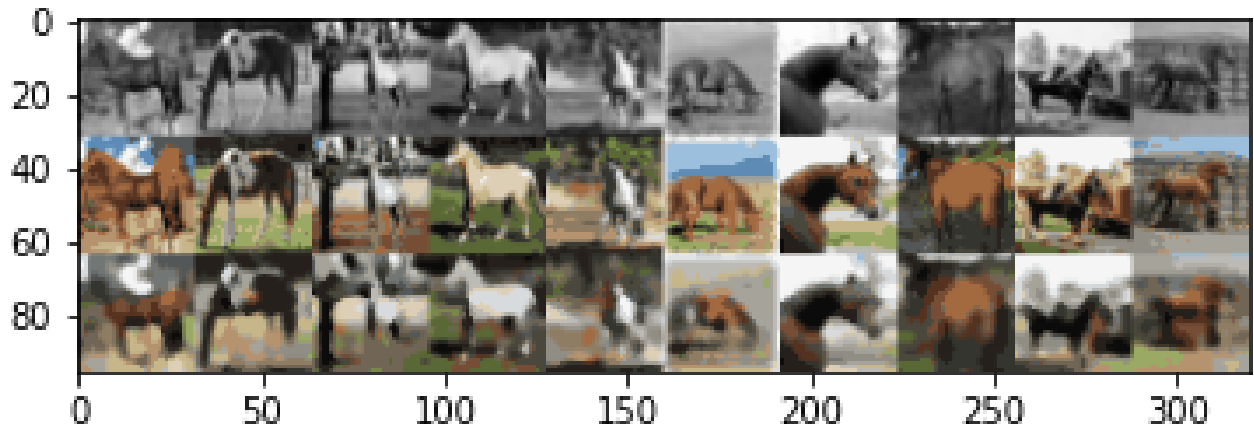
```

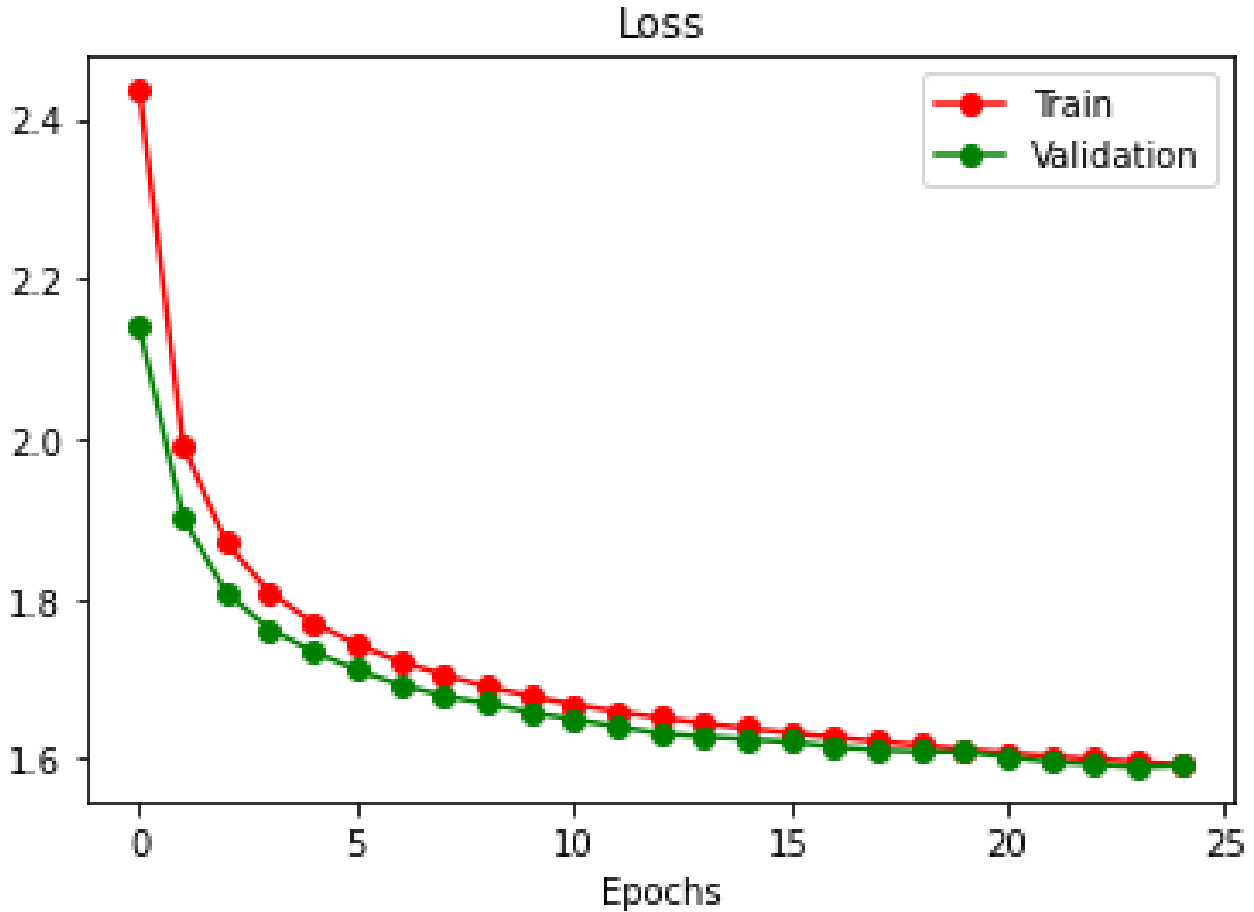

b blue[0.5pt]

Run main training loop of `PoolUpsampleNet`. This will train the CNN for a few epochs using the cross-entropy objective. It will generate some images showing the trained result at the end. Do these results look good to you? Why or why not?

Answer:

The model accuracy is 41.1%, means that 41.1% of the image area is correctly colored. It seems a good result.





c blue[1.0pt]

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.

Note:

1. Please ignore biases when answering the questions.
2. Please ignore `nn.BatchNorm2d` when answering the number of weights, outputs and connections, but we still accept answers that do.

Hint:

1. `nn.Upsample` does not have parameters (this will help you answer the number of weights).
2. Think about when the input width and height are both doubled, how will the dimension of feature maps in each layer change? If you know this, you will know how dimension scaling will affect the number of outputs and connections.

Answer:

when each input dimension (width/height) is not doubled

Total number of weights: $NIC * NF * k^2 + 4 NF^2 * k^2 + NF * NC * k^2 + NC^2 * k^2$

Total number of outputs: $2880NF + 3328NC$

Total number of connects: $1024 * k^2 * NIC * NF + 640 k^2 * NF^2 + 256 * k^2 * NF^2 + 1024 k^2 * NC^2$

when each input dimension (width/height) is doubled

the number of weights would not change, but the number of outputs and connects would be multiplied by 4.

5 Strided and Transposed Dilated Convolutions [2 pts]

For this part, instead of using `nn.MaxPool2d` layers to reduce the dimensionality of the tensors, we will increase the step size of the preceding `nn.Conv2d` layers, and instead of using `nn.Upsample` layers to increase the dimensionality of the tensors, we will use *transposed* convolutions. Transposed convolutions aim to apply the same operations as convolutions but in the opposite direction. For example, while increasing the stride from 1 to 2 in a convolution forces the filters to skip over every other position as they slide across the input tensor, increasing the stride from 1 to 2 in a transposed convolution adds “empty” space around each element of the input tensor, as if reversing the skipping over every other position done by the convolution. We will be using a **dilation rate of 1** for the transposed convolution. Excellent visualizations of convolutions and transposed convolutions have been developed by [?] and can be found on their GitHub page⁴.

a blue[0.5pt]

Complete the model `ConvTransposeNet`, following the diagram in Figure ?? . Use the PyTorch layers `nn.Conv2d`, `nn.ReLU`, `nn.BatchNorm2d` and `nn.ConvTranspose2d`. As before, your CNN should be configurable by parameters `kernel`, `dilation`, `num_in_channels`, `num_filters`, and `num_colours`. Use the following parameterizations (if not specified, assume default parameters):

- `nn.Conv2d`: The number of input and output filters, and the kernel size, should be set in the same way as Section 4. For the first two `nn.Conv2d` layers, set `stride` to 2 and set `padding` to 1.
- `nn.BatchNorm2d`: The number of features should be specified in the same way as for Section 4.
- `nn.ConvTranspose2d`: The number of input filters should match the second dimension of the *input* tensor. The number of output filters should match the second dimension of the *output* tensor. Set `kernel_size` to parameter `kernel`. Set `stride` to 2, set `dilation` to 1, and set both `padding` and `output_padding` to 1.

Answer:

⁴https://github.com/vdumoulin/conv_arithmetic

```

output_padding = 1

##### YOUR CODE GOES HERE #####
self.first = nn.Sequential(
    nn.Conv2d(in_channels=num_in_channels, out_channels=num_filters,
              kernel_size=kernel, stride=2, padding=1),
    nn.BatchNorm2d(num_features=num_filters),
    nn.ReLU(),
)

self.second = nn.Sequential(
    nn.Conv2d(in_channels=num_filters, out_channels=num_filters * 2,
              kernel_size=kernel, stride=2, padding=1),
    nn.BatchNorm2d(num_features=num_filters * 2),
    nn.ReLU(),
)

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

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

self.last = nn.Conv2d(in_channels=num_filters * 2, out_channels=num_filters * 2,
                      kernel_size=kernel, padding=padding)
#####

def forward(self, x):
    ##### YOUR CODE GOES HERE #####
    x_1 = self.first(x)
    x_2 = self.second(x_1)
    x_3 = self.third(x_2)
    x_4 = self.forth(x_3)
    return self.last(x_4)
#####

```

chatGPT:

chatGPT can give the nearly accurate answer based on the information given in prompt.

< 2 / 2 >

S

For this part, instead of using `nn.MaxPool2d` layers to reduce the dimensionality of the tensors, we will increase the step size of the preceding `nn.Conv2d` layers, and instead of using `nn.Upsample` layers to increase the dimensionality of the tensors, we will use transposed convolutions. Transposed convolutions aim to apply the same operations as convolutions but in the opposite direction. For example, while increasing the stride from 1 to 2 in a convolution forces the filters to skip over every other position as they slide across the input tensor, increasing the stride from 1 to 2 in a transposed convolution adds “empty” space around each element of the input tensor, as if reversing the skipping over every other position done by the convolution. We will be using a dilation rate of 1 for the transposed convolution. Complete the model `ConvTransposeNet`, following the diagram in Figure 1b. Use the PyTorch layers `nn.Conv2d`, `nn.ReLU`, `nn.BatchNorm2d` and `nn.ConvTranspose2d`. As before, your CNN should be configurable by parameters `kernel`, `dilation`, `num in channels`, `num filters`, and `num colours`. Use the following parameterizations (if not specified, assume default parameters):

- `nn.Conv2d`: The number of input and output filters, and the kernel size, should be set in the same way as Section 4. For the first two `nn.Conv2d` layers, set stride to 2 and set padding to 1.
- `nn.BatchNorm2d`: The number of features should be specified in the same way as for Section 4.
- `nn.ConvTranspose2d`: The number of input filters should match the second dimension of the input tensor. The number of output filters should match the second dimension of the output tensor. Set kernel size to parameter `kernel`. Set stride to 2, set dilation to 1, and set both padding and output padding to 1.





python

Copy code

```
import torch.nn as nn

class ConvTransposeNet(nn.Module):
    def __init__(self, kernel=3, dilation=1, num_in_channels=3, num_filters=64, num_colours=10):
        super(ConvTransposeNet, self).__init__()
        self.conv1 = nn.Conv2d(num_in_channels, num_filters, kernel_size=kernel, stride=dilation)
        self.bn1 = nn.BatchNorm2d(num_filters)
        self.relu1 = nn.ReLU(inplace=True)
        self.conv2 = nn.Conv2d(num_filters, num_filters*2, kernel_size=kernel, stride=dilation)
        self.bn2 = nn.BatchNorm2d(num_filters*2)
        self.relu2 = nn.ReLU(inplace=True)
        self.conv3 = nn.ConvTranspose2d(num_filters*2, num_filters, kernel_size=kernel, stride=dilation)
        self.bn3 = nn.BatchNorm2d(num_filters)
        self.relu3 = nn.ReLU(inplace=True)
        self.conv4 = nn.ConvTranspose2d(num_filters, num_colours, kernel_size=kernel, stride=dilation)

    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
```

b blue[0.5pt]

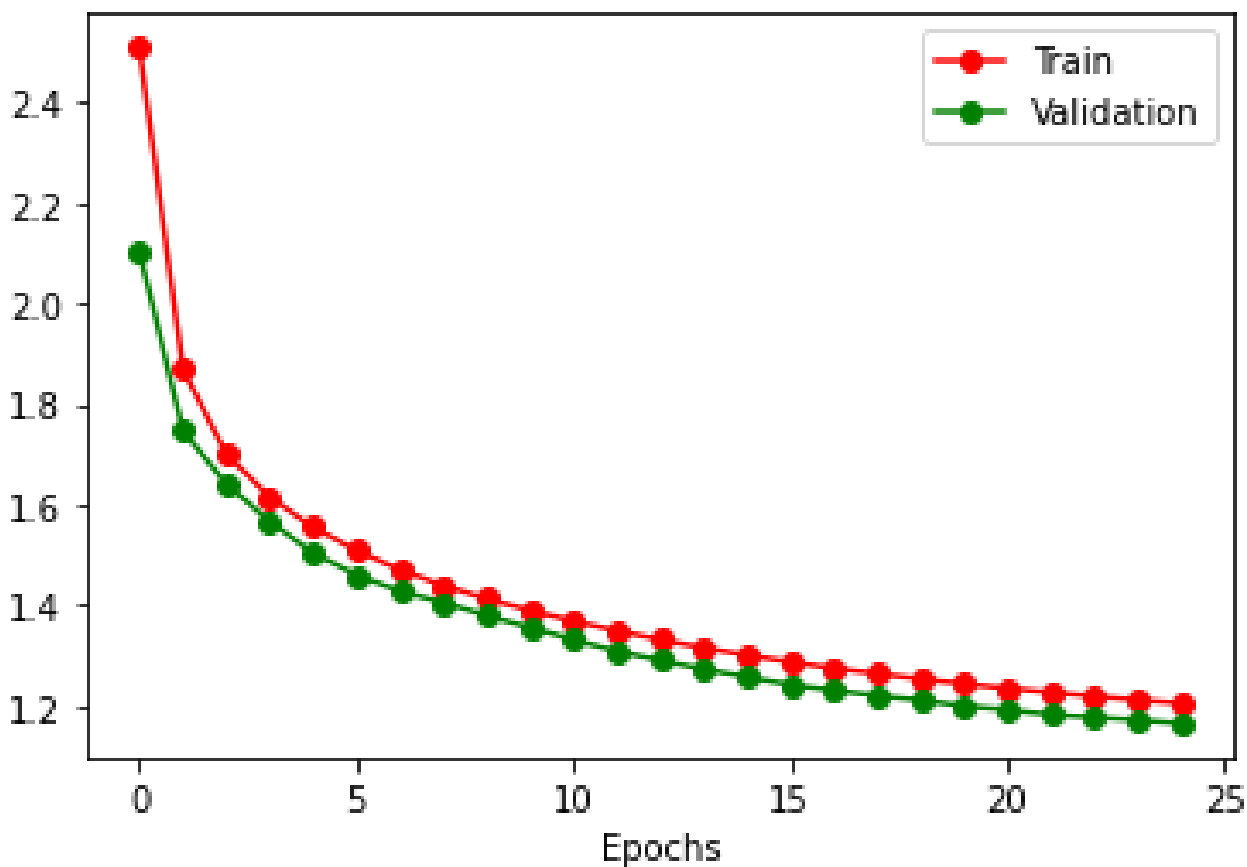
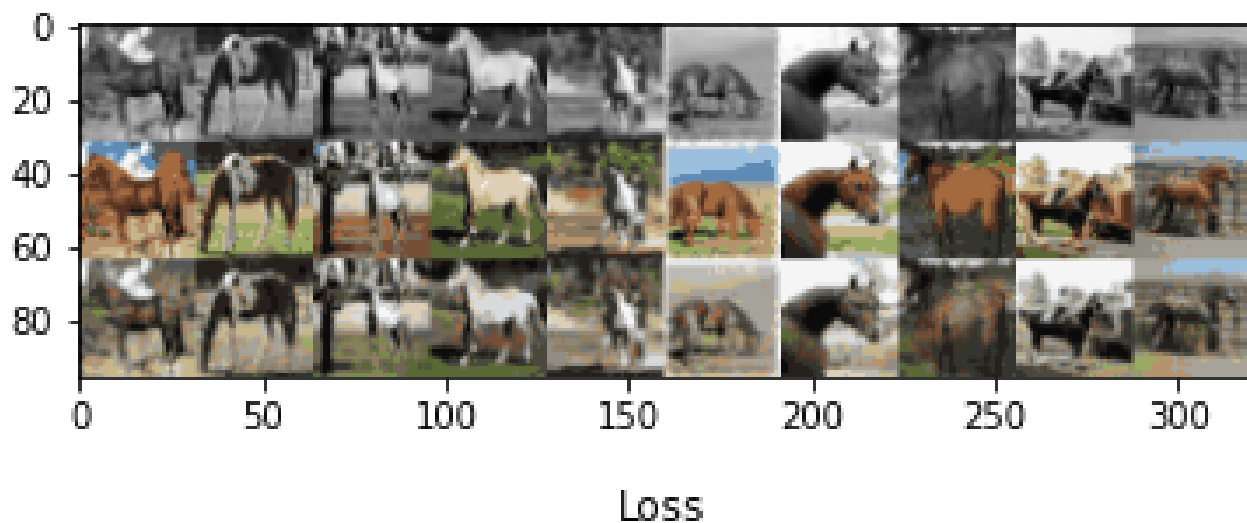
Train the model for at least 25 epochs using a batch size of 100 and a kernel size of 3. Plot the training curve, and include this plot in your write-up.

Answer:

```

Epoch [4/25], Val Loss: 1.5666, Val Acc: 42.0%, Time(s): 3.53
Epoch [5/25], Loss: 1.5551, Time (s): 4
Epoch [5/25], Val Loss: 1.5036, Val Acc: 44.1%, Time(s): 4.46
Epoch [6/25], Loss: 1.5083, Time (s): 5
Epoch [6/25], Val Loss: 1.4598, Val Acc: 45.4%, Time(s): 5.37
Epoch [7/25], Loss: 1.4705, Time (s): 6
Epoch [7/25], Val Loss: 1.4282, Val Acc: 46.1%, Time(s): 6.32
Epoch [8/25], Loss: 1.4392, Time (s): 7
Epoch [8/25], Val Loss: 1.4048, Val Acc: 46.6%, Time(s): 7.29
Epoch [9/25], Loss: 1.4125, Time (s): 8
Epoch [9/25], Val Loss: 1.3793, Val Acc: 47.4%, Time(s): 8.30
Epoch [10/25], Loss: 1.3892, Time (s): 9
Epoch [10/25], Val Loss: 1.3556, Val Acc: 48.2%, Time(s): 9.36
Epoch [11/25], Loss: 1.3683, Time (s): 10
Epoch [11/25], Val Loss: 1.3311, Val Acc: 49.1%, Time(s): 10.44
Epoch [12/25], Loss: 1.3490, Time (s): 11
Epoch [12/25], Val Loss: 1.3084, Val Acc: 49.9%, Time(s): 11.53
Epoch [13/25], Loss: 1.3313, Time (s): 12
Epoch [13/25], Val Loss: 1.2902, Val Acc: 50.4%, Time(s): 12.68
Epoch [14/25], Loss: 1.3151, Time (s): 13
Epoch [14/25], Val Loss: 1.2725, Val Acc: 51.1%, Time(s): 13.85
Epoch [15/25], Loss: 1.3003, Time (s): 14
Epoch [15/25], Val Loss: 1.2568, Val Acc: 51.6%, Time(s): 15.35
Epoch [16/25], Loss: 1.2868, Time (s): 16
Epoch [16/25], Val Loss: 1.2415, Val Acc: 52.1%, Time(s): 16.78
Epoch [17/25], Loss: 1.2745, Time (s): 17
Epoch [17/25], Val Loss: 1.2303, Val Acc: 52.5%, Time(s): 18.08
Epoch [18/25], Loss: 1.2632, Time (s): 18
Epoch [18/25], Val Loss: 1.2198, Val Acc: 52.8%, Time(s): 19.40
Epoch [19/25], Loss: 1.2529, Time (s): 20
Epoch [19/25], Val Loss: 1.2108, Val Acc: 53.0%, Time(s): 20.73
Epoch [20/25], Loss: 1.2433, Time (s): 21
Epoch [20/25], Val Loss: 1.1995, Val Acc: 53.4%, Time(s): 22.10
Epoch [21/25], Loss: 1.2345, Time (s): 23
Epoch [21/25], Val Loss: 1.1918, Val Acc: 53.6%, Time(s): 23.51
Epoch [22/25], Loss: 1.2262, Time (s): 24
Epoch [22/25], Val Loss: 1.1824, Val Acc: 54.0%, Time(s): 24.96
Epoch [23/25], Loss: 1.2186, Time (s): 25
Epoch [23/25], Val Loss: 1.1768, Val Acc: 54.2%, Time(s): 26.43
Epoch [24/25], Loss: 1.2114, Time (s): 27
Epoch [24/25], Val Loss: 1.1714, Val Acc: 54.3%, Time(s): 28.42
Epoch [25/25], Loss: 1.2047, Time (s): 29
Epoch [25/25], Val Loss: 1.1645, Val Acc: 54.5%, Time(s): 29.98

```



c blue[0.5pt]

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?

Answer:

Results is better than section 4. The validation accuracy is 54.5% which is higher than 41.1%. Since section 4 uses maxpool to reduce the width and height of the images, so there would be fewer parameters to learn

for next layer. Therefore, by using upsample in this model, we allow more features to be learnt, thus has a better result.

d blue[0.5pt]

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 (assuming we want to maintain the shapes of all tensors shown in Figure ??)? Note: PyTorch documentation for `nn.Conv2d`⁵ and `nn.ConvTranspose2d`⁶ includes equations that can be used to calculate the shape of the output tensors given the parameters.

Answer:

kernel size is 4, padding is 1 and *output_padding* is 0.

kernel size is 5 padding is 2 and *output_padding* is 1.

e blue[0pt]

Re-train a few more `ConvTransposeNet` models using different batch sizes (e.g., 32, 64, 128, 256, 512) with a fixed number of epochs. Describe the effect of batch sizes on the training/validation loss, and the final image output quality. You do *not* need to attach the final output images.

6 Skip Connections

A skip connection in a neural network is a connection which skips one or more layer and connects to a later layer. We will introduce skip connections to the model we implemented in Section 5.

a blue[0.5pt]

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 (see Figure ??). This type of skip-connection is introduced by [?], and is called a “UNet”. Following the `ConvTransposeNet` class that you have completed, complete the `__init__` and `forward` methods of the `UNet` class in Section 6 of the notebook. Hint: You will need to use the function `torch.cat`.

Answer:

⁵<https://pytorch.org/docs/stable/generated/torch.nn.Conv2d.html>

⁶<https://pytorch.org/docs/stable/generated/torch.nn.ConvTranspose2d.html>

```

##### YOUR CODE GOES HERE #####
self.first = nn.Sequential(
    nn.Conv2d(in_channels=num_in_channels, out_channels=num_filters,
              kernel_size=kernel, stride=2, padding=1),
    nn.BatchNorm2d(num_features=num_filters),
    nn.ReLU(),
)

self.second = nn.Sequential(
    nn.Conv2d(in_channels=num_filters, out_channels=num_filters * 2,
              kernel_size=kernel, stride=2, padding=1),
    nn.BatchNorm2d(num_features=num_filters * 2),
    nn.ReLU(),
)

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

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

self.last = 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 #####
    x1 = self.first(x)
    x2 = self.second(x1)
    x3 = self.third(x2)
    x4 = self.forth(torch.cat([x3, x1], dim=1))
    return self.last(torch.cat([x4, x], dim=1))
#####

```

chatGPT:

chatGPT can give the nearly accurate answer based on the information given in prompt.

s

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 (see Figure 1c). This type of skip-connection is introduced by Ronneberger et al.

[2015], and is called a “UNet”. Following the ConvTransposeNet class that you have completed,

complete the init and forward methods of the UNet class in Section 6 of the notebook. Hint: You will need to use the function torch.cat.



Here is the updated implementation of the UNet class:



python

Copy code

```
import torch.nn as nn

class UNet(nn.Module):
    def __init__(self, kernel=3, dilation=1, num_in_channels=3, num_filters=64, num_colours=3):
        super(UNet, self).__init__()
        self.downconv1 = nn.Conv2d(num_in_channels, num_filters, kernel_size=kernel, dilation=dilation)
        self.downbn1 = nn.BatchNorm2d(num_filters)
        self.downrelu1 = nn.ReLU(inplace=True)
        self.downconv2 = nn.Conv2d(num_filters, num_filters*2, kernel_size=kernel, dilation=dilation)
        self.downbn2 = nn.BatchNorm2d(num_filters*2)
        self.downrelu2 = nn.ReLU(inplace=True)
        self.upconv1 = nn.ConvTranspose2d(num_filters*2, num_filters, kernel_size=kernel, dilation=dilation)
        self.upbn1 = nn.BatchNorm2d(num_filters)
        self.uprelu1 = nn.ReLU(inplace=True)
        self.upconv2 = nn.ConvTranspose2d(num_filters, num_colours, kernel_size=kernel, dilation=dilation)

    def forward(self, x):
        x1 = self.downconv1(x)
        x1 = self.downbn1(x1)
        x1 = self.downrelu1(x1)
        x2 = self.downconv2(x1)
        x2 = self.downbn2(x2)
        x2 = self.downrelu2(x2)
        x = self.upconv1(x2)
        x = self.upbn1(x)
        x = self.uprelu1(x)
        x = self.upconv2(x)
        x1 = nn.functional.interpolate(x1, scale_factor=4, mode='bilinear', align_corners=True)
        x = torch.cat([x, x1], dim=1)
        return x
```

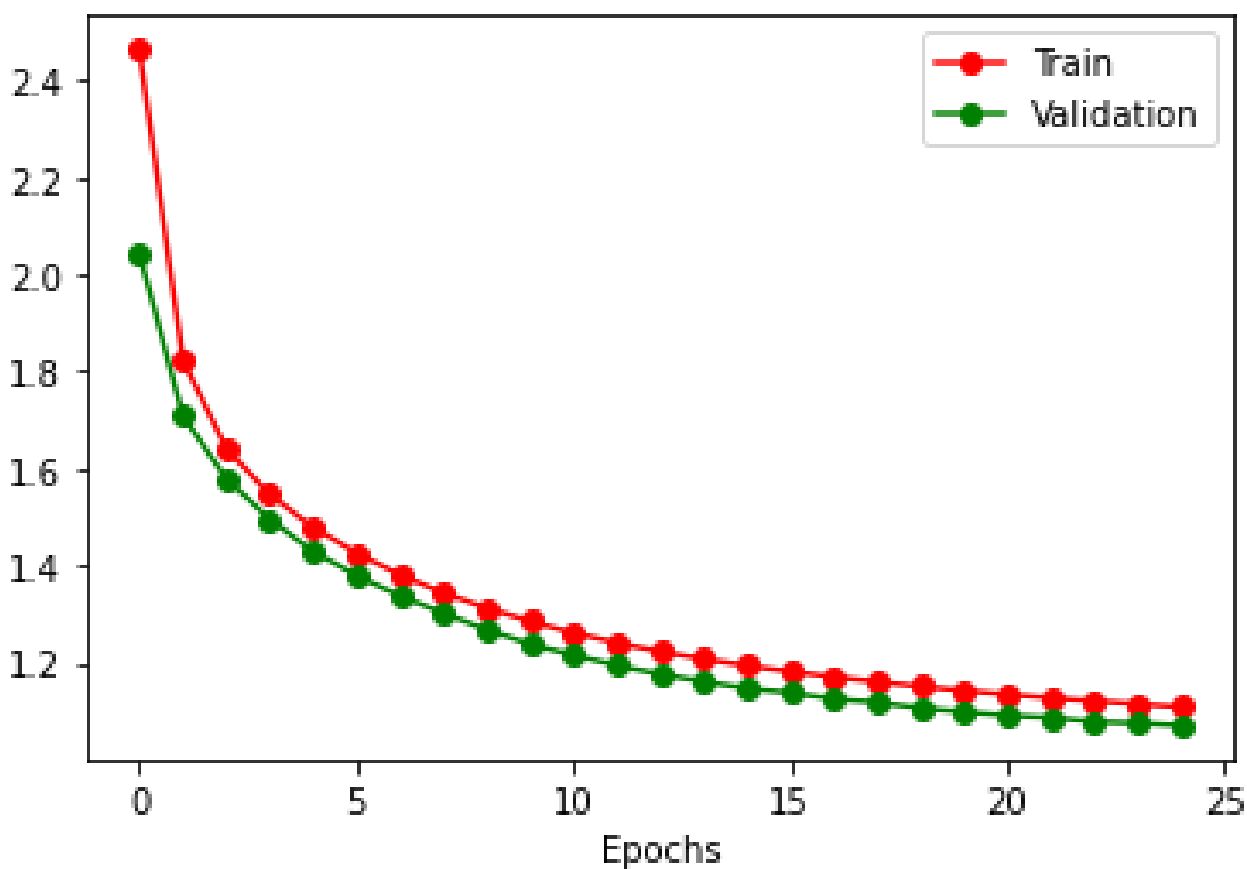
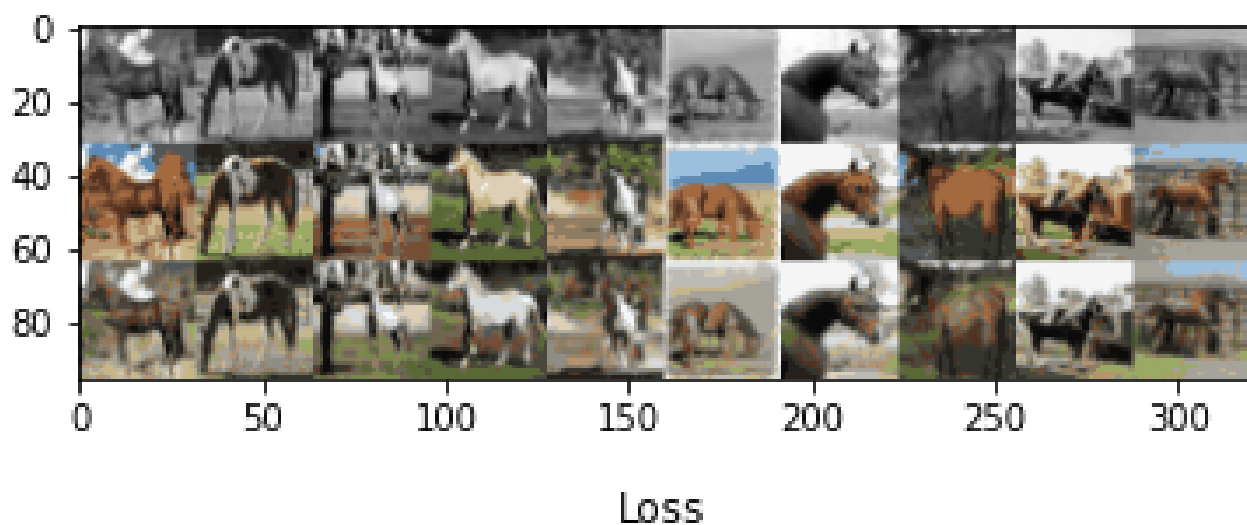
b **blue[0.5pt]**

Train the model for at least 25 epochs using a batch size of 100 and a kernel size of 3. Plot the training curve, and include this plot in your write-up.

Answer:

Beginning training ...

Epoch [1/25], Loss: 2.4641, Time (s): 0
Epoch [1/25], Val Loss: 2.0446, Val Acc: 31.9%, Time(s): 0.94
Epoch [2/25], Loss: 1.8246, Time (s): 1
Epoch [2/25], Val Loss: 1.7094, Val Acc: 39.2%, Time(s): 1.74
Epoch [3/25], Loss: 1.6417, Time (s): 2
Epoch [3/25], Val Loss: 1.5798, Val Acc: 42.4%, Time(s): 2.58
Epoch [4/25], Loss: 1.5472, Time (s): 3
Epoch [4/25], Val Loss: 1.4963, Val Acc: 44.9%, Time(s): 3.45
Epoch [5/25], Loss: 1.4799, Time (s): 4
Epoch [5/25], Val Loss: 1.4298, Val Acc: 47.0%, Time(s): 4.49
Epoch [6/25], Loss: 1.4263, Time (s): 5
Epoch [6/25], Val Loss: 1.3804, Val Acc: 48.5%, Time(s): 5.61
Epoch [7/25], Loss: 1.3821, Time (s): 6
Epoch [7/25], Val Loss: 1.3385, Val Acc: 49.9%, Time(s): 6.61
Epoch [8/25], Loss: 1.3449, Time (s): 7
Epoch [8/25], Val Loss: 1.3027, Val Acc: 50.9%, Time(s): 7.64
Epoch [9/25], Loss: 1.3131, Time (s): 8
Epoch [9/25], Val Loss: 1.2688, Val Acc: 52.1%, Time(s): 8.74
Epoch [10/25], Loss: 1.2859, Time (s): 9
Epoch [10/25], Val Loss: 1.2387, Val Acc: 53.2%, Time(s): 9.85
Epoch [11/25], Loss: 1.2621, Time (s): 10
Epoch [11/25], Val Loss: 1.2166, Val Acc: 53.8%, Time(s): 10.96
Epoch [12/25], Loss: 1.2416, Time (s): 11
Epoch [12/25], Val Loss: 1.1946, Val Acc: 54.6%, Time(s): 12.13
Epoch [13/25], Loss: 1.2238, Time (s): 12
Epoch [13/25], Val Loss: 1.1769, Val Acc: 55.0%, Time(s): 13.32
Epoch [14/25], Loss: 1.2082, Time (s): 14
Epoch [14/25], Val Loss: 1.1615, Val Acc: 55.4%, Time(s): 14.55
Epoch [15/25], Loss: 1.1945, Time (s): 15
Epoch [15/25], Val Loss: 1.1478, Val Acc: 55.8%, Time(s): 15.81
Epoch [16/25], Loss: 1.1822, Time (s): 16
Epoch [16/25], Val Loss: 1.1389, Val Acc: 56.0%, Time(s): 17.43
Epoch [17/25], Loss: 1.1712, Time (s): 18
Epoch [17/25], Val Loss: 1.1261, Val Acc: 56.4%, Time(s): 18.92
Epoch [18/25], Loss: 1.1611, Time (s): 19
Epoch [18/25], Val Loss: 1.1188, Val Acc: 56.5%, Time(s): 20.30
Epoch [19/25], Loss: 1.1518, Time (s): 21
Epoch [19/25], Val Loss: 1.1073, Val Acc: 57.0%, Time(s): 21.69
Epoch [20/25], Loss: 1.1432, Time (s): 22
Epoch [20/25], Val Loss: 1.0994, Val Acc: 57.1%, Time(s): 23.13
Epoch [21/25], Loss: 1.1353, Time (s): 24
Epoch [21/25], Val Loss: 1.0922, Val Acc: 57.3%, Time(s): 24.60
Epoch [22/25], Loss: 1.1281, Time (s): 25
Epoch [22/25], Val Loss: 1.0878, Val Acc: 57.3%, Time(s): 26.13
Epoch [23/25], Loss: 1.1216, Time (s): 27
Epoch [23/25], Val Loss: 1.0792, Val Acc: 57.6%, Time(s): 27.67
Epoch [24/25], Loss: 1.1155, Time (s): 28
Epoch [24/25], Val Loss: 1.0766, Val Acc: 57.6%, Time(s): 29.58
Epoch [25/25], Loss: 1.1098, Time(s): 30
Epoch [25/25], Val Loss: 1.0713, Val Acc: 57.7%, Time(s): 31.39



c blue[1.0pt]

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.

Answer:

The result increased validation accuracy to 57.7%, decrease validation loss to 1.0691. It improves the final

image output quality.

Here is the reasons. First, skip connection may add additional trainable parameters. Second, skip connection could save more information compared to pooling or strided convolution.