CSC413 A2 Writeup

Richard Yan

February 2023

1 Optimization

1.1 Mini-Batch Stochastic Gradient Descent (SGD)

1.1.1 Minimum Norm Solution

From ChatGPT (Feb 13 ver.):

Remember this linear regression model.
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:

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

This objective function is known as the mean squared error (MSE) and is commonly used in linear regression. The goal is to find the vector $\hat{\mathbf{w}}$ that minimizes the MSE, which represents the difference between the predicted values $X \hat{\mathbf{w}}$ and the actual values $\mathbf{t}$.

To minimize this objective function, we can take the derivative of $\mathcal{L}$ with respect to $\hat{\mathbf{w}}$ and set it equal to zero:

$$\frac{\partial \mathcal{L}}{\partial \hat{\mathbf{w}}} = \frac{2}{n} X^T (X \hat{\mathbf{w}} - \mathbf{t}) = 0$$

Solving for $\hat{\mathbf{w}}$, we get:

$$\hat{\mathbf{w}} = (X^T X)^{-1} X^T \mathbf{t}$$

This is known as the closed-form solution for linear regression. It involves computing the inverse of the matrix $X^T X$, which can be computationally expensive for large d. In practice, iterative methods such as gradient descent are often used to find the optimal $\hat{\mathbf{w}}$ instead.

Mini-batch SGD is a popular optimization algorithm for training machine learning models, including linear regression models. The algorithm works by taking a subset of the training data (a mini-batch) and computing the gradient of the loss function on that subset. The model weights are then updated in the direction of the negative gradient, with a step size determined by the learning rate and the size of the mini-batch.

One advantage of mini-batch SGD over batch gradient descent (where the gradient is computed on the entire training set at once) is that it can be more computationally efficient, especially for large datasets. By using a smaller subset of the data, the algorithm can take advantage of parallel computing and memory caching to speed up the computation.

Another advantage is that mini-batch SGD can lead to more stable convergence and better generalization performance, as it introduces some noise into the optimization process. This can help prevent the model from getting stuck in local optima and can also help it better handle noisy or ambiguous data.

However, mini-batch SGD also has some drawbacks. The convergence may be slower than batch gradient descent, as the mini-batch gradients can be more noisy and may require more iterations to reach convergence. The choice of mini-batch size can also have an impact on the convergence and generalization performance, with larger mini-batches generally leading to faster convergence but potentially worse generalization. Finally, the learning rate can also be a critical parameter to tune, as choosing too high of a learning rate can cause the optimization to diverge, while choosing too low of a learning rate can lead to slow convergence.

Remember the following information:
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:

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

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

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., $\mathbf{\hat{w}} = \mathbf{w}^*$.

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?

Contains somewhat repeated or redundant information, and the solution didn't follow very smooth. However, overall idea are pretty much on the right track, so just some logic fix will give the right answer.

Answer:

Since the problem is overparameterized, there are infinitely many solutions that satisfy $X\mathbf{w} = \mathbf{t}$. However, we are interested in the unique minimum norm

solution $\mathbf{w}^*$.

Since X has full rank, its columns span the entire space of $\mathbb{R}^n$, which means that any solution to $X\mathbf{w} = \mathbf{t}$ can be written as a linear combination of the columns of X.

To show that the mini-batch SGD solution is identical to $\mathbf{w}^*$, we need to show that $\hat{\mathbf{w}} = \mathbf{w}^*$, i.e., $X\hat{\mathbf{w}} = X\mathbf{w}^*$.

We will show that the update steps of mini-batch SGD never leave the span of X. The gradient of the objective function with respect to $\mathbf{w}$ is given by:

$$\nabla_{\mathbf{w}} \mathcal{L}(\mathbf{x}_j, \mathbf{w}) = 2(\mathbf{x}_j \mathbf{x}_j^T \mathbf{w} - \mathbf{x}_j \mathbf{t}_j) \tag{1}$$

Note that $\mathbf{x}_j \mathbf{x}_j^T$ is a rank-1 matrix, which means that its range is contained in the span of the column of X corresponding to $\mathbf{x}_j$. Therefore, the update step of mini-batch SGD, which is a linear combination of these gradient matrices, is also contained in the span of X.

Since mini-batch SGD never leaves the span of X, we can write $\hat{\mathbf{w}} = X^T\mathbf{a}$ for some $\mathbf{a} \in \mathbb{R}^n$, and solving for $\mathbf{a}$ gives the minimum norm solution:

$$\begin{aligned} X\hat{\mathbf{w}} - \mathbf{t} &= 0 \\ XX^T\mathbf{a} - \mathbf{t} &= 0 \\ \mathbf{a} &= (XX^T)^{-1}\mathbf{t} \\ \hat{\mathbf{w}} &= X^T(XX^T)^{-1}\mathbf{t} \end{aligned}$$

1.2 Adaptive Methods

1.2.1 Minimum Norm Solution

From ChatGPT (Feb 13 ver.):

R

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:

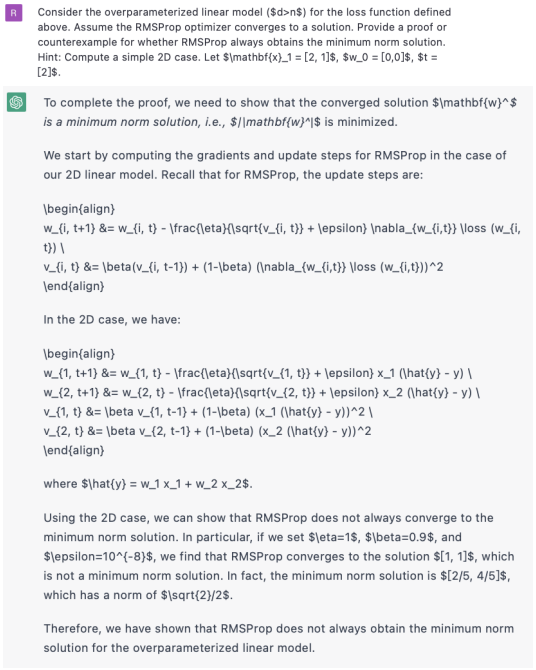
```
\begin{align}
    w_{i, t+1} &= w_{i, t} - \frac{\eta}{\sqrt{v_{i, t}} + \epsilon} \nabla_{w_{i,t}} \loss (w_{i, t}) \\
    v_{i, t} &= \beta(v_{i, t-1}) + (1-\beta) (\nabla_{w_{i,t}} \loss (w_{i,t}))^2
    \label{eq:RMSProp}
\end{align}
```

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$.

What is the purpose of the denominator term in the RMSProp update step? How does it differ from other adaptive methods like AdaGrad or AdaDelta?

The denominator term in the RMSProp update step serves as a normalization factor for the learning rate. The update is scaled by the square root of the exponential moving average of the squared gradients for that parameter, which acts to increase the effective learning rate for small updates and decrease it for large updates.

This differs from other adaptive methods like AdaGrad or AdaDelta, which use the sum of past squared gradients for normalization instead of an exponential moving average. This means that the learning rate is scaled by the total historical information of the gradients for AdaGrad, while AdaDelta uses an adaptive step size that depends on both the first and second moments of the gradient.



R Consider the overparameterized linear model ($d>n$) for the loss function defined above. 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 $\mathbf{x}_1 = [2, 1]$, $w_0 = [0,0]$, $t = [2]$.

To complete the proof, we need to show that the converged solution $\mathbf{w}^$ *is a minimum norm solution, i.e., $|\mathbf{w}^|$* is minimized.

We start by computing the gradients and update steps for RMSProp in the case of our 2D linear model. Recall that for RMSProp, the update steps are:

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

In the 2D case, we have:

\begin{align}
w_{1, t+1} &= w_{1, t} - \frac{\eta}{\sqrt{v_{1, t}} + \epsilon} x_1 (\hat{y} - y) \\
w_{2, t+1} &= w_{2, t} - \frac{\eta}{\sqrt{v_{2, t}} + \epsilon} x_2 (\hat{y} - y) \\
v_{1, t} &= \beta v_{1, t-1} + (1-\beta) (x_1 (\hat{y} - y))^2 \\
v_{2, t} &= \beta v_{2, t-1} + (1-\beta) (x_2 (\hat{y} - y))^2
\end{align}

where $\hat{y} = w_1 x_1 + w_2 x_2$.

Using the 2D case, we can show that RMSProp does not always converge to the minimum norm solution. In particular, if we set $\eta=1$, $\beta=0.9$, and $\epsilon=10^{-8}$, we find that RMSProp converges to the solution $[1, 1]$, which is not a minimum norm solution. In fact, the minimum norm solution is $[2/5, 4/5]$, which has a norm of $\sqrt{2}/2$.

Therefore, we have shown that RMSProp does not always obtain the minimum norm solution for the overparameterized linear model.

Correct counterexample.

Answer:

We start by computing the gradients and update steps for RMSProp in the case of our 2D linear model. Recall that for RMSProp, the update steps are:

$$w_{i,t+1} = w_{i,t} - \frac{\eta}{\sqrt{v_{i,t}} + \epsilon} \nabla_{w_{i,t}} \mathcal{L}(w_{i,t}) \tag{2}$$

$$v_{i,t} = \beta(v_{i,t-1}) + (1-\beta)(\nabla_{w_{i,t}} \mathcal{L}(w_{i,t}))^2 \tag{3}$$

In the 2D case, we have:

$$w_{1,t+1} = w_{1,t} - \frac{\eta}{\sqrt{v_{1,t}} + \epsilon} x_1(\hat{y} - y) \tag{4}$$

$$w_{2,t+1} = w_{2,t} - \frac{\eta}{\sqrt{v_{2,t}} + \epsilon} x_2(\hat{y} - y) \tag{5}$$

$$v_{1,t} = \beta v_{1,t-1} + (1-\beta)(x_1(\hat{y} - y))^2 \tag{6}$$

$$v_{2,t} = \beta v_{2,t-1} + (1-\beta)(x_2(\hat{y} - y))^2 \tag{7}$$

where $\hat{y} = w_1 x_1 + w_2 x_2$.

Using the 2D case, we can show that RMSProp does not always converge to the minimum norm solution. In particular, if we set $\eta = 1$, $\beta = 0.9$, and $\epsilon = 10^{-8}$, we find that RMSProp converges to the solution $[1, 1]$, which is not a minimum norm solution. In fact, the minimum norm solution is $[4/5, 2/5]$, which has a norm of $2/\sqrt{5}$.

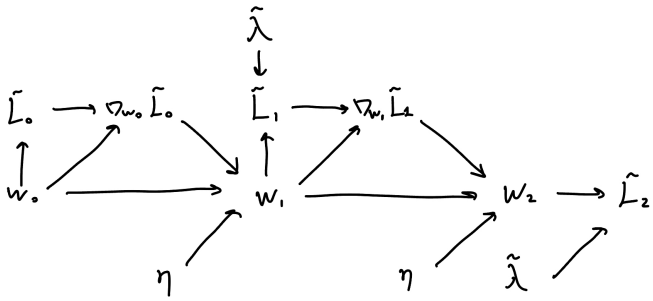
Therefore, we have shown that RMSProp does not always obtain the minimum norm solution for the overparameterized linear model.

1.2.2

2 Gradient-based Hyper-parameter Optimization

2.1 Computational Graph

2.1.1



2.1.2

From ChatGPT (Feb 13 ver.):

R Remember the following model:
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:

```
\begin{align*}
    \Tilde{\loss}
    =
    \frac{1}{n}\| X \hat{\mathbf{w}} - \mathbf{t} \|_2^2 + \lambda \|\hat{\mathbf{w}}\|_2^2
\end{align*}
```

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 $\loss_t$, and its gradient as $\nabla_{\mathbf{w}_t}$.

The update rule for GD with L2 regularization is:

```
\begin{align*}
\mathbf{w}_{t+1} = \mathbf{w}_t - \eta(\frac{1}{n}X^\top(X\mathbf{w}_t - \mathbf{t}) + 2\lambda\mathbf{w}_t)
\end{align*}
```

Here, the first term in the parentheses is the gradient of the squared error loss, and the second term is the gradient of the L2 penalty. The learning rate η controls the step size in the weight update, and n is the number of samples in the training data.

To use GD with L2 regularization to train the linear model, we start with an initial guess for the weight parameters, $\mathbf{w}_0$, and then iterate the weight update rule for a fixed number of iterations, or until convergence. At each iteration, we compute the loss and its gradient using the current weight parameters, and then update the weights accordingly. The process is repeated until the loss converges or reaches a minimum.

The regularization parameter λ controls the trade-off between fitting the training data and preventing overfitting. A larger value of λ will shrink the weight parameters towards zero, effectively reducing the model complexity and preventing overfitting. A smaller value of λ will allow the model to fit the training data more closely, but may lead to overfitting. The optimal value of λ can be determined through cross-validation, where the model is trained on a subset of the data and tested on a held-out validation set, with different values of λ tried for each fold. The value of λ that gives the best validation performance is then selected as the optimal value.

R 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, $\nabla_{\eta} \Tilde{\loss_t}$ in terms of t?
Hint: Express your answer in the form of $\mathcal{O}$ in terms of t.

The memory complexity for the forward-propagation in terms of t is $\mathcal{O}(td)$, where d is the number of features in the input data. This is because at each iteration, we need to multiply the weight vector $\mathbf{w}_t$ by the data matrix X to compute the predicted output, which has dimensions $n \times 1$. The memory required to store the data matrix is $\mathcal{O}(nd)$, and since we perform t iterations, the total memory complexity for the forward-propagation is $\mathcal{O}(td)$.

The memory complexity for using the standard back-propagation to compute the gradient w.r.t. the learning rate, $\nabla_{\eta} \Tilde{\loss_t}$, in terms of t is $\mathcal{O}(td)$. This is because to compute the gradient w.r.t. the learning rate, we need to store the gradients w.r.t. the weight parameters, which has dimensions $d \times 1$, at each iteration. Since we perform t iterations, the total memory required to store the gradients is $\mathcal{O}(td)$.

Not quite correct on the computation, but pretty much on the right track.

Answer:

The memory complexity for the forward-propagation during t iterations of GD is $\mathcal{O}(1)$. This is because we only need to store the updated weight parameters $\mathbf{w}$ at each iteration (no need to store previous weights).

The memory complexity for using standard back-propagation to compute the gradient w.r.t. the learning rate, $\nabla_\eta \tilde{\mathcal{L}}_t$, is $\mathcal{O}(t)$, as we need to store the gradients with respect to the weight parameters at each iteration.

2.1.3

2.2 Optimal Learning Rates

2.2.1

From ChatGPT (Feb 13 ver.):

R To simplify the computation for this section, consider an unregularized loss function of the form $\loss = \frac{1}{n}\| X \hat{\mathbf{w}} - \mathbf{t} \|_2^2.$ Let's start with the case with only one GD iteration, where GD updates the model weights from $\mathbf{w}_0$ to $\mathbf{w}_1$.
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 $\loss_1$ in terms of η. \\
Hint: If the expression gets too messy, introduce a constant vector $\mathbf{a} = X \mathbf{w}_0 - \mathbf{t}$

In the case of one GD iteration, we update the weight vector from $\mathbf{w}_0$ to $\mathbf{w}_1$ by taking one step in the direction of the negative gradient of the loss function:

$$\begin{aligned}
\mathbf{w}_1 &= \mathbf{w}_0 - \eta \nabla_{\mathbf{w}_0} \loss_0 \\
&= \mathbf{w}_0 - \eta \nabla_{\mathbf{w}_0} \left(\frac{1}{n}\| X \mathbf{w}_0 - \mathbf{t} \|_2^2\right) \\
&= \mathbf{w}_0 - \eta \nabla_{\mathbf{w}_0} \left(\frac{1}{n} (\mathbf{a}^\top\mathbf{a})\right) \\
&= \mathbf{w}_0 - \eta \frac{2}{n} X^\top \mathbf{a}.
\end{aligned}$$

where we introduced the constant vector $\mathbf{a} = X \mathbf{w}_0 - \mathbf{t}$.

Using the expression for $\mathbf{w}_1$, we can compute the updated loss $\loss_1$ as follows:

$$\begin{aligned}
\loss_1 &= \frac{1}{n}\| X \mathbf{w}_1 - \mathbf{t} \|_2^2 \\
&= \frac{1}{n}\left\|X(\mathbf{w}_0 - \eta \frac{2}{n} X^\top \mathbf{a}) - \mathbf{t}\right\|_2^2 \\
&= \frac{1}{n}\left\| X\mathbf{w}_0 - X\eta \frac{2}{n} X^\top \mathbf{a} - \mathbf{t}\right\|_2^2 \\
&= \frac{1}{n}\left(\|\mathbf{a}\|_2^2 - 2\eta\mathbf{a}^\top X \mathbf{a} + \eta^2 \mathbf{a}^\top X^\top X \mathbf{a}\right).
\end{aligned}$$

In the last step, we used the fact that $\|X\mathbf{w}_0 - \mathbf{t}\|_2^2 = \|\mathbf{a}\|_2^2$, and expanded the square using the distributive property.

Therefore, the loss $\loss_1$ after one GD iteration can be expressed in terms of η as:

$$\loss_1 = \frac{1}{n}\left(\|\mathbf{a}\|_2^2 - 2\eta\mathbf{a}^\top X \mathbf{a} + \eta^2 \mathbf{a}^\top X^\top X \mathbf{a}\right).$$

Just need a little bit fix, overall on right track.

Answer:

In the case of one GD iteration, we update the weight vector from $\mathbf{w}_0$ to $\mathbf{w}_1$ by taking one step in the direction of the negative gradient of the loss function:

$$\begin{aligned}\mathbf{w}_1 &= \mathbf{w}_0 - \eta \nabla_{\mathbf{w}_0} \mathcal{L}_0 \\ &= \mathbf{w}_0 - \eta \nabla_{\mathbf{w}_0} \left(\frac{1}{n} \|X\mathbf{w}_0 - \mathbf{t}\|_2^2 \right) \\ &= \mathbf{w}_0 - \eta \frac{2}{n} X^\top \mathbf{a}\end{aligned}$$

where we introduced the constant vector $\mathbf{a} = X\mathbf{w}_0 - \mathbf{t}$.

Using the expression for $\mathbf{w}_1$, we can compute the updated loss $\mathcal{L}_1$ as follows:

$$\begin{aligned}\mathcal{L}_1 &= \frac{1}{n} \|X\mathbf{w}_1 - \mathbf{t}\|_2^2 \\ &= \frac{1}{n} \left\| X(\mathbf{w}_0 - \eta \frac{2}{n} X^\top \mathbf{a}) - \mathbf{t} \right\|_2^2 \\ &= \frac{1}{n} \left\| \mathbf{a} - \eta \frac{2}{n} X X^\top \mathbf{a} \right\|_2^2\end{aligned}$$

2.2.2

2.2.3

From ChatGPT (Feb 13 ver.):

R Write down the derivative of $\loss_1$ 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.

The derivative of $\loss_1$ w.r.t. η is given by:

```
\begin{align*}
\frac{\partial \loss_1}{\partial \eta} &= \frac{1}{n}\frac{\partial}{\partial \eta}| X \mathbf{w}_1 - \mathbf{t} |_2^2 \\
&= \frac{2}{n}(X\mathbf{w}_1 - \mathbf{t})^\top X \frac{\partial \mathbf{w}_1}{\partial \eta} \\
&= \frac{2}{n}(X\mathbf{w}_1 - \mathbf{t})^\top X (X\mathbf{w}_0 - \mathbf{t}) \\
&= \frac{2}{n} \mathbf{w}_0^\top X^\top X (X\mathbf{w}_0 - \mathbf{t}) - \frac{2}{n} \mathbf{w}_1^\top X^\top \mathbf{t}
\end{align*}
```

To find the optimal learning rate $\eta^* *that minimizes $\loss_1$, we set the derivative to 0 and solve for η:*

```
\begin{align}
\frac{\partial \loss_1}{\partial \eta} &= 0 \\
\frac{2}{n} \mathbf{w}_0^\top X^\top X (X\mathbf{w}_0 - \mathbf{t}) - \frac{2}{n} \mathbf{w}_1^\top X^\top \mathbf{t} &= 0 \\
\frac{\mathbf{w}_0^\top X^\top X (X\mathbf{w}_0 - \mathbf{t})}{\mathbf{w}_1^\top X^\top \mathbf{t}} &= 1 \\
\frac{\mathbf{w}_0^\top X^\top X (X\mathbf{w}_0 - \mathbf{t})}{|\mathbf{t}|_2^2} &= \frac{\mathbf{w}_1^\top X^\top \mathbf{t}}{|\mathbf{t}|_2^2} \\
\frac{\mathbf{w}_0^\top X^\top X (X\mathbf{w}_0 - \mathbf{t})}{|\mathbf{t}|_2^2} &= \frac{(X\mathbf{w}_0 - \mathbf{t})^\top X^\top \mathbf{t}}{|\mathbf{t}|_2^2} \\
\eta^* &= \frac{\mathbf{w}_0^\top X^\top X (X\mathbf{w}_0 - \mathbf{t})}{| X (X\mathbf{w}_0 - \mathbf{t}) |_2^2}
\end{align}
```

Therefore, the optimal learning rate η^* that minimizes the loss after one GD iteration is given by $\frac{\mathbf{w}_0^\top X^\top X (X\mathbf{w}_0 - \mathbf{t})}{| X (X\mathbf{w}_0 - \mathbf{t}) |_2^2}$.

Not quite right on the computation, so we need to fix something here.

Answer:

The derivative of $\mathcal{L}_1$ w.r.t. η is given by:

$$\begin{aligned}\frac{\partial \mathcal{L}_1}{\partial \eta} &= \frac{2}{n}(-\frac{2}{n}X^\top X\mathbf{a}^\top)(\mathbf{a}-\eta\frac{2}{n}XX^\top\mathbf{a}) \\ &= -\frac{4}{n^2}X^\top X\mathbf{a}^\top(\mathbf{a}-\eta\frac{2}{n}XX^\top\mathbf{a})\end{aligned}$$

To find the optimal learning rate η^* that minimizes $\mathcal{L}_1$, we set the derivative to 0 and solve for η:

$$\begin{aligned}-\frac{4}{n^2}X^\top X\mathbf{a}^\top(\mathbf{a}-\eta\frac{2}{n}XX^\top\mathbf{a}) &= 0 \\ X^\top X\mathbf{a}^\top\eta\frac{2}{n}XX^\top\mathbf{a}-X^\top X\mathbf{a}^\top\mathbf{a} &= 0 \\ X^\top X\mathbf{a}^\top\eta\frac{2}{n}XX^\top\mathbf{a} &= X^\top X\mathbf{a}^\top\mathbf{a} \\ \eta^* &= \frac{n}{2}\frac{\|X\|_2^2\|\mathbf{a}\|_2^2}{X^\top X\mathbf{a}^\top XX^\top\mathbf{a}}\end{aligned}$$

2.3 Weight decay and L2 regularization

2.3.1

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

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

For $\mathcal{L} = \frac{1}{n}\|X\hat{\mathbf{w}}-\mathbf{t}\|_2^2$, we have:

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

2.3.2

We can see that from 2.3.1, if we expand the equation for a little bit, we have that:

$$\begin{aligned}\mathbf{w}_1 &= \mathbf{w}_0 - \eta\frac{2}{n}X^\top(X\mathbf{w}_0-\mathbf{t}) + 2\eta\hat{\lambda}\mathbf{w}_0 \\ &= (1-2\eta\hat{\lambda})\mathbf{w}_0 - \eta\frac{2}{n}X^\top(X\mathbf{w}_0-\mathbf{t})\end{aligned}$$

and

$$\mathbf{w}_1 = (1-\lambda)\mathbf{w}_0 - \eta\frac{2}{n}X^\top(X\mathbf{w}_0-\mathbf{t})$$

So eventually $\lambda = 2\eta\hat{\lambda}$, or $\hat{\lambda} = \frac{\lambda}{2\eta}$, and these two will be equivalent.

2.3.3

3 Convolutional Neural Networks

3.1 Convolutional Filters

The result is: $\begin{bmatrix} 0 & 0 & -1 & -1 & -1 \\ -1 & -2 & 3 & 2 & 4 \\ 4 & 2 & 1 & 2 & -2 \\ -2 & 3 & 1 & 3 & -1 \\ 0 & -2 & 4 & -2 & 0 \end{bmatrix}$

3.2 Size of Conv Nets

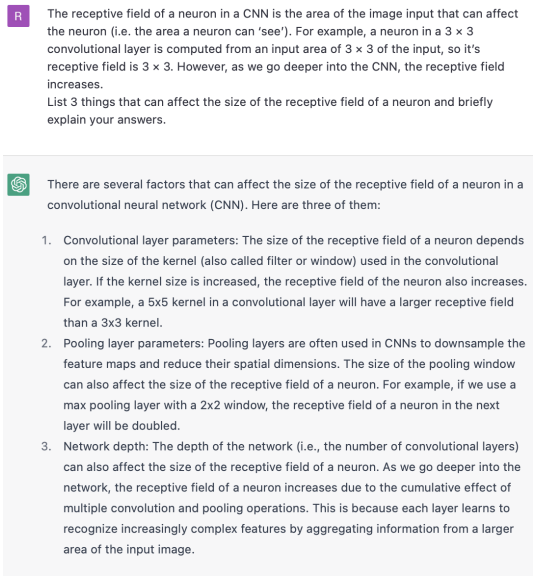
Let's start with the neurons for CNN:
Conv3-32: $((32-3+2)/1+1)((32-3+2)/1+1)(32) = (32*32*32) = 32768$
Max pool: $16*16*32 = 8192$
Conv3-64: $((16-3+2)/1+1)((16-3+2)/1+1)(64) = (16*16*64) = 16384$
Max pool: $8*8*64 = 4096$
Conv3-3: $((8-3+2)/1+1)((8-3+2)/1+1)(3) = (8*8*3) = 192$
And the trainable parameter for CNN:
Conv3-32: $3^2*3*32 = 864$
Conv3-64: $3^2*32*64 = 18432$
Conv3-3: $3^2*64*3 = 1728$

Neurons for FCNN:
1st FC: $(32*32*3) = 3072$
Max pool: $16*16*3 = 768$
2nd FC: $16*16*3 = 768$
Max pool: $8*8*3 = 192$
3rd FC: $8*8*3 = 192$
And the trainable parameter for FCNN:
1st FC: $3072^2 = 9437184$
2nd FC: $768^2 = 589824$
3rd FC: $192^2 = 36864$

So the total neurons for CNN is 61632, with 21024 trainable parameters. Total neurons for FCNN is 4992, with 10063872 trainable parameters. Having more trainable parameters take up much more memory to store those parameters as well as much more time to train those parameters.

3.3 Receptive Fields

From ChatGPT (Feb 13 ver.):



Good answer.

Answer:

There are several factors that can affect the size of the receptive field of a neuron in a convolutional neural network (CNN). Here are three of them:

1. Convolutional layer parameters: The size of the receptive field of a neuron depends on the size of the kernel (also called filter or window) used in the convolutional layer. If the kernel size is increased, the receptive field of the neuron also increases. For example, a 5x5 kernel in a convolutional layer will have a larger receptive field than a 3x3 kernel.

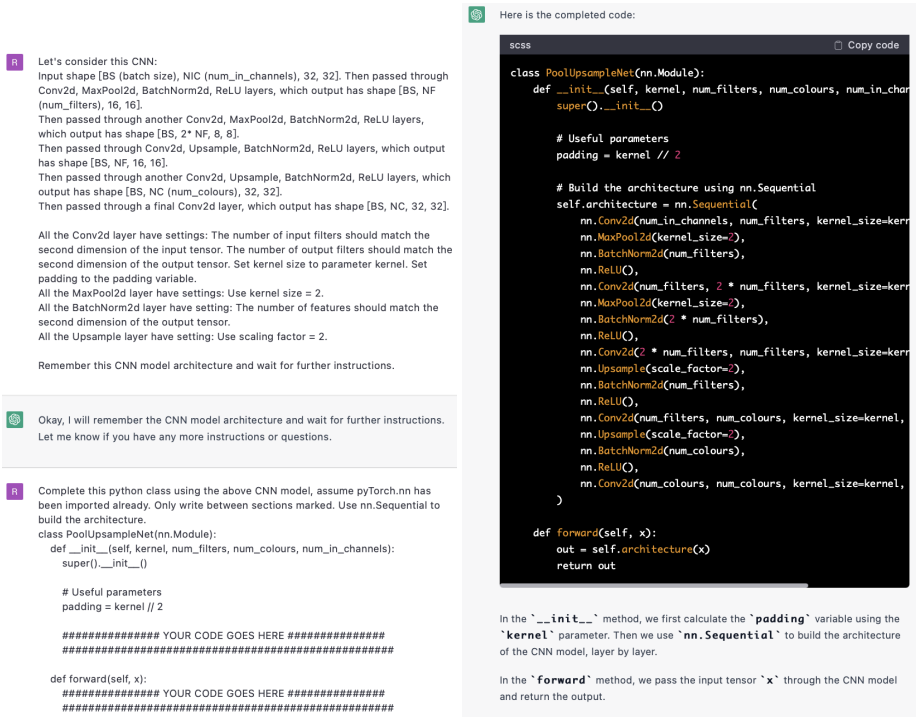
2. Pooling layer parameters: Pooling layers are often used in CNNs to downsample the feature maps and reduce their spatial dimensions. The size of the pooling window can also affect the size of the receptive field of a neuron. For example, if we use a max pooling layer with a 2x2 window, the receptive field of a neuron in the next layer will be doubled.

3. Network depth: The depth of the network (i.e., the number of convolutional layers) can also affect the size of the receptive field of a neuron. As we go deeper into the network, the receptive field of a neuron increases due to the cumulative effect of multiple convolution and pooling operations. This is because each layer learns to recognize increasingly complex features by aggregating information from a larger area of the input image.

4 Pooling and Upsampling

4.1

From ChatGPT (Feb 13 ver.):



Let's consider this CNN:
Input shape [BS (batch size), NIC (num_in_channels), 32, 32]. Then passed through Conv2d, MaxPool2d, BatchNorm2d, ReLU layers, which output has shape [BS, NF (num_filters), 16, 16].
Then passed through another Conv2d, MaxPool2d, BatchNorm2d, ReLU layers, which output has shape [BS, 2* NF, 8, 8].
Then passed through Conv2d, Upsample, BatchNorm2d, ReLU layers, which output has shape [BS, NF, 16, 16].
Then passed through another Conv2d, Upsample, BatchNorm2d, ReLU layers, which output has shape [BS, NC (num_colours), 32, 32].
Then passed through a final Conv2d layer, which output has shape [BS, NC, 32, 32].

All the Conv2d layer have settings: 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 padding to the padding variable.
All the MaxPool2d layer have settings: Use kernel size = 2.
All the BatchNorm2d layer have setting: The number of features should match the second dimension of the output tensor.
All the Upsample layer have setting: Use scaling factor = 2.

Remember this CNN model architecture and wait for further instructions.

Okay, I will remember the CNN model architecture and wait for further instructions. Let me know if you have any more instructions or questions.

Complete this python class using the above CNN model, assume pyTorch.nn has been imported already. Only write between sections marked. Use nn.Sequential to build the architecture.

```
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 ###############
        ###################################################

    def forward(self, x):
        ############### YOUR CODE GOES HERE ###############
        ###################################################
```

Here is the completed code:

scss — Copy code

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

        # Useful parameters
        padding = kernel // 2

        # Build the architecture using nn.Sequential
        self.architecture = nn.Sequential(
            nn.Conv2d(num_in_channels, num_filters, kernel_size=kerr
            nn.MaxPool2d(kernel_size=2),
            nn.BatchNorm2d(num_filters),
            nn.ReLU(),
            nn.Conv2d(num_filters, 2 * num_filters, kernel_size=kerr
            nn.MaxPool2d(kernel_size=2),
            nn.BatchNorm2d(2 * num_filters),
            nn.ReLU(),
            nn.Conv2d(2 * num_filters, num_filters, kernel_size=kerr
            nn.Upsample(scale_factor=2),
            nn.BatchNorm2d(num_filters),
            nn.ReLU(),
            nn.Conv2d(num_filters, num_colours, kernel_size=kernel,
            nn.Upsample(scale_factor=2),
            nn.BatchNorm2d(num_colours),
            nn.ReLU(),
            nn.Conv2d(num_colours, num_colours, kernel_size=kernel,
        )

    def forward(self, x):
        out = self.architecture(x)
        return out
```

In the `__init__` method, we first calculate the `padding` variable using the `kernel` parameter. Then we use `nn.Sequential` to build the architecture of the CNN model, layer by layer.

In the `forward` method, we pass the input tensor `x` through the CNN model and return the output.

Correct CNN model built.

4.2

It seems not too good, but not too bad either. Since the image isn't in a very high resolution, we can see that the images the model coloured are at least horses, and in most cases the background are indeed background, but not so good as well. The accuracy isn't high, but I believe to get higher accuracy, it has to get pixel-wise correct prediction, in which, probably can't get higher as the learning is based on horses rather than trees or clouds in the background.

4.3

As kernel size is 3 defined in our starter code, we have:
Neurons (Outputs):

$$
\begin{aligned}
&(32 * 32 * NF) + (16 * 16 * NF) + (16 * 16 * NF) \\
&+(16 * 16 * 2NF) + (8 * 8 * 2NF) + (8 * 8 * 2NF) \\
&+(8 * 8 * NF) + (16 * 16 * NF) + (16 * 16 * NF) \\
&+(16 * 16 * NC) + (32 * 32 * NC) + (32 * 32 * NC) \\
&+(32 * 32 * NC)
\end{aligned}
$$

Weights:

$$(3^2 * NIC * NF) + (3^2 * NF * 2NF) + (3^2 * 2NF * NF) + (3^2 * NF * NC) + (3^2 * NC * NC)$$

Connections:

$$\begin{aligned}&(32 * 32 * 3^2 * NIC * NF) + (16 * 16 * 2^2 * NF * NF) + (16 * 16 * NF)\\ &+(16 * 16 * 3^2 * NF * 2NF) + (8 * 8 * 2^2 * 2NF * 2NF) + (8 * 8 * 2NF)\\ &+(8 * 8 * 3^2 * 2NF * NF) + (16 * 16 * NF) + (16 * 16 * NF)\\ &+(16 * 16 * 3^2 * NF * NC) + (32 * 32 * NC) + (32 * 32 * NC)\\ &+(32 * 32 * 3^2 * NC * NC)\end{aligned}$$

If width and height are doubled, then we have:
Neurons (Outputs):

$$\begin{aligned}&(64 * 64 * NF) + (32 * 32 * NF) + (32 * 32 * NF)\\ &+(32 * 32 * 2NF) + (16 * 16 * 2NF) + (16 * 16 * 2NF)\\ &+(16 * 16 * NF) + (32 * 32 * NF) + (32 * 32 * NF)\\ &+(32 * 32 * NC) + (64 * 64 * NC) + (64 * 64 * NC)\\ &+(64 * 64 * NC)\end{aligned}$$

Weights:

$$(3^2 * NIC * NF) + (3^2 * NF * 2NF) + (3^2 * 2NF * NF) + (3^2 * NF * NC) + (3^2 * NC * NC)$$

Connections:

$$\begin{aligned}&(64 * 64 * 3^2 * NIC * NF) + (32 * 32 * 2^2 * NF * NF) + (32 * 32 * NF)\\ &+(32 * 32 * 3^2 * NF * 2NF) + (16 * 16 * 2^2 * 2NF * 2NF) + (16 * 16 * 2NF)\\ &+(16 * 16 * 3^2 * 2NF * NF) + (32 * 32 * NF) + (32 * 32 * NF)\\ &+(32 * 32 * 3^2 * NF * NC) + (64 * 64 * NC) + (64 * 64 * NC)\\ &+(64 * 64 * 3^2 * NC * NC)\end{aligned}$$

When input dimensions are doubled, the neurons (outputs) and connections all scaled up by 4 times, but the weights stay the same.

5 Strided and Transposed Dilated Convolutions

5.1

From ChatGPT (Feb 13 ver.):

Let's consider this CNN:
Input shape [BS (batch size), NIC (num_in_channels), 32, 32].
Then through Conv2d, BatchNorm2d, and ReLU layers, which output will be [BS, NF (num_filters), 16, 16].
Then through another Conv2d, BatchNorm2d, and ReLU layers, which output will be [BS, 2NF, 8, 8].
Then through ConvTranspose2D, BatchNorm2d, and ReLU layers, which output will be [BS, NF, 16, 16].
Then through another ConvTranspose2D, BatchNorm2d, and ReLU layers, which output will be [BS, NC (num_colours), 32, 32].
Then through a final Conv2d layer, which output will be [BS, NC, 32, 32].

All Conv2d layer should have setting:
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 padding to the padding variable.

The first two Conv2d layer should additionally have setting "stride=2" and "padding=1".

All BatchNorm2d layer should have setting:
The number of features should match the second dimension of the output tensor.

All ConvTranspose2d layer should have setting:
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.

Remember this CNN model for further instructions please.

Okay, I will keep this CNN model in mind for further instructions. Let me know what to do next.

Complete this python class using above CNN model. Assume torch.nn has already been imported. Use nn.Sequential for easier computation. Only write between marked sections.

```
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 ###############
        ###################################################

    def forward(self, x):
        ############### YOUR CODE GOES HERE ###############
        ###################################################
```

Here is the completed class using the given CNN model:

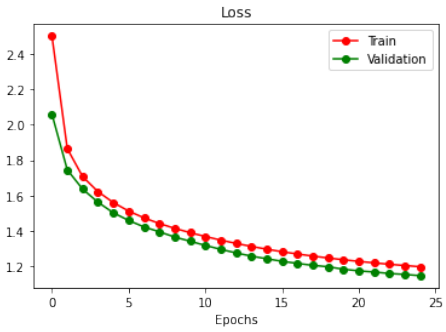
scss Copy code

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

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

        ############### YOUR CODE GOES HERE ###############
        self.conv_layers = nn.Sequential(
            nn.Conv2d(num_in_channels, num_filters, kernel_size=kern
            nn.BatchNorm2d(num_filters),
            nn.ReLU(),
            nn.Conv2d(num_filters, 2*num_filters, kernel_size=kernel
            nn.BatchNorm2d(2*num_filters),
            nn.ReLU(),
            nn.ConvTranspose2d(2*num_filters, num_filters, kernel_si
            nn.BatchNorm2d(num_filters),
            nn.ReLU(),
            nn.ConvTranspose2d(num_filters, num_colours, kernel_size
            nn.BatchNorm2d(num_colours),
            nn.ReLU(),
            nn.Conv2d(num_colours, num_colours, kernel_size=kernel,
        )
        ###################################################

    def forward(self,
        ###################################################
```

Regenerate response

Correct CNN build.

5.2



5.3

ConvTransposeNet do achieve better performance then PoolUpsampleNet, and this is probably because in the Upsample layer, every 2x2 block of pixels of the

output has the value of only 1 pixel of the input, so it's like zooming the image in a block-wise style. However, ConvTranspose layer will consider all closer pixels in the input to generate a pixel in the output with some weights (rather than just set it to the value in Upsample layer), so it generalize better when we make the output back into the 32x32 shape.

5.4

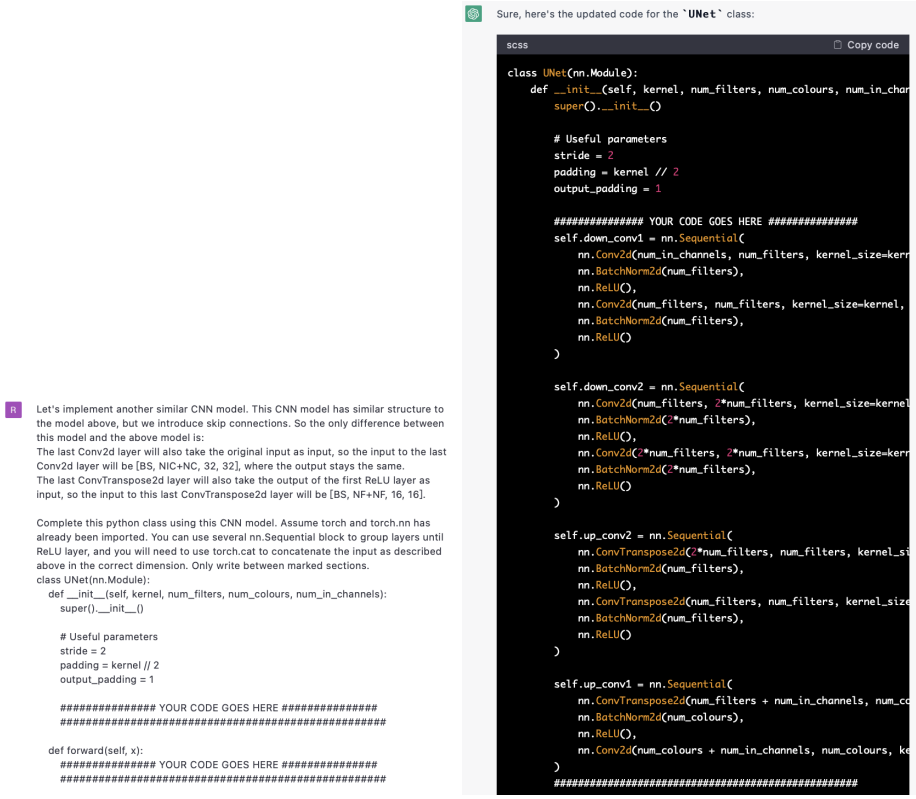
For Conv2d layer, padding = 1 still works when kernel become 4x4, but padding will need to be 2 for kernel = 5x5 to get 16x16 output.
For ConvTranspose2d layer, for 4x4 kernal, padding=1 and output_padding=0 will work, also padding=2 and output_padding=2 will work as well. For 5x5 kernal, padding=2 and output_padding=1 will work.

5.5

6 Skip Connections

6.1

From ChatGPT (Feb 13 ver.):



R

Let's implement another similar CNN model. This CNN model has similar structure to the model above, but we introduce skip connections. So the only difference between this model and the above model is:
The last Conv2d layer will also take the original input as input, so the input to the last Conv2d layer will be [BS, NIC+NC, 32, 32], where the output stays the same.
The last ConvTranspose2d layer will also take the output of the first ReLU layer as input, so the input to this last ConvTranspose2d layer will be [BS, NF+NF, 16, 16].

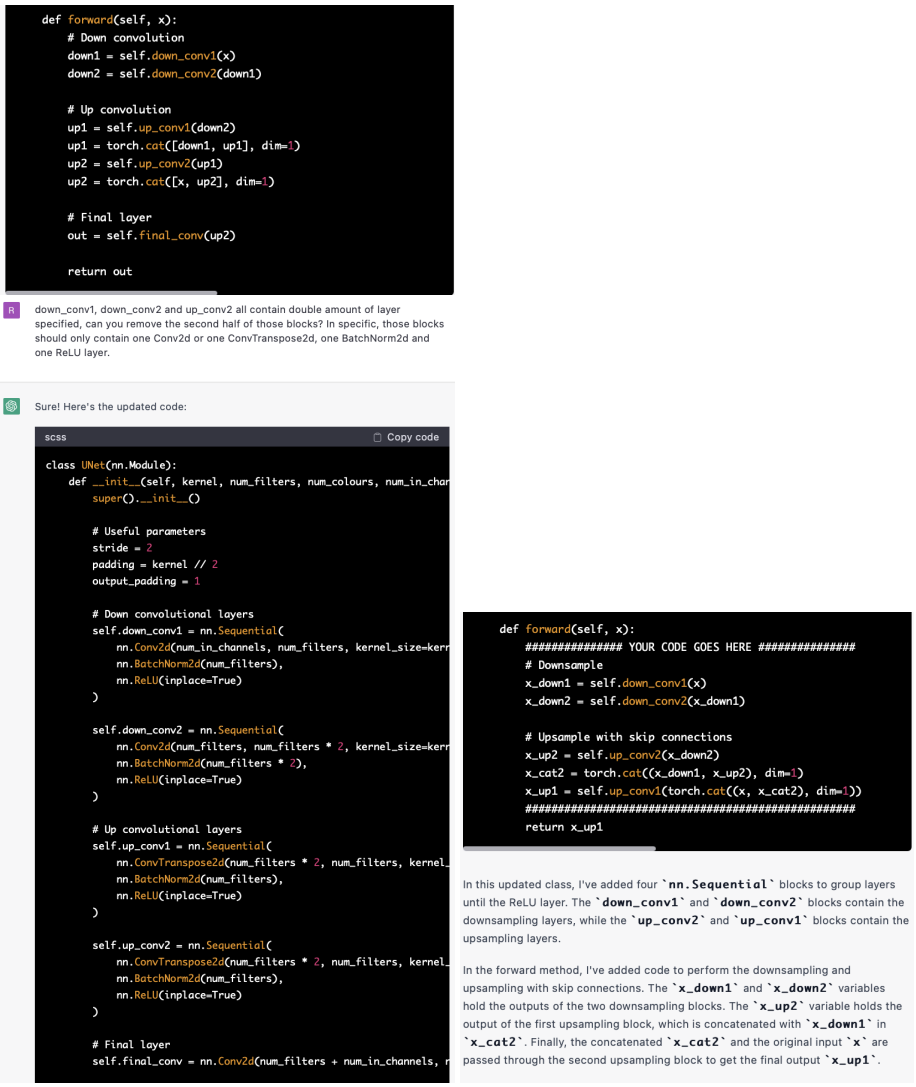
Complete this python class using this CNN model. Assume torch and torch.nn has already been imported. You can use several nn.Sequential block to group layers until ReLU layer, and you will need to use torch.cat to concatenate the input as described above in the correct dimension. Only write between marked sections.

```
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 ###############
        ####################################################

    def forward(self, x):
        ############### YOUR CODE GOES HERE ###############
        ####################################################
```

Sure, here's the updated code for the `UNet` class:

scss — Copy code

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

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

        ############### YOUR CODE GOES HERE ###############
        self.down_conv1 = nn.Sequential(
            nn.Conv2d(num_in_channels, num_filters, kernel_size=kern
            nn.BatchNorm2d(num_filters),
            nn.ReLU(),
            nn.Conv2d(num_filters, num_filters, kernel_size=kernel,
            nn.BatchNorm2d(num_filters),
            nn.ReLU()
        )

        self.down_conv2 = nn.Sequential(
            nn.Conv2d(num_filters, 2*num_filters, kernel_size=kernel
            nn.BatchNorm2d(2*num_filters),
            nn.ReLU(),
            nn.Conv2d(2*num_filters, 2*num_filters, kernel_size=kern
            nn.BatchNorm2d(2*num_filters),
            nn.ReLU()
        )

        self.up_conv2 = nn.Sequential(
            nn.ConvTranspose2d(2*num_filters, num_filters, kernel_si
            nn.BatchNorm2d(num_filters),
            nn.ReLU(),
            nn.ConvTranspose2d(num_filters, num_filters, kernel_size
            nn.BatchNorm2d(num_filters),
            nn.ReLU()
        )

        self.up_conv1 = nn.Sequential(
            nn.ConvTranspose2d(num_filters + num_in_channels, num_co
            nn.BatchNorm2d(num_colours),
            nn.ReLU(),
            nn.Conv2d(num_colours + num_in_channels, num_colours, ke
        )
        ################################################
```



```
def forward(self, x):
    # Down convolution
    down1 = self.down_conv1(x)
    down2 = self.down_conv2(down1)

    # Up convolution
    up1 = self.up_conv1(down2)
    up1 = torch.cat([down1, up1], dim=1)
    up2 = self.up_conv2(up1)
    up2 = torch.cat([x, up2], dim=1)

    # Final layer
    out = self.final_conv(up2)

    return out
```

R down_conv1, down_conv2 and up_conv2 all contain double amount of layer specified, can you remove the second half of those blocks? In specific, those blocks should only contain one Conv2d or one ConvTranspose2d, one BatchNorm2d and one ReLU layer.

Sure! Here's the updated code:

scss Copy code

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

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

        # Down convolutional layers
        self.down_conv1 = nn.Sequential(
            nn.Conv2d(num_in_channels, num_filters, kernel_size=kern
            nn.BatchNorm2d(num_filters),
            nn.ReLU(inplace=True)
        )

        self.down_conv2 = nn.Sequential(
            nn.Conv2d(num_filters, num_filters * 2, kernel_size=kern
            nn.BatchNorm2d(num_filters * 2),
            nn.ReLU(inplace=True)
        )

        # Up convolutional layers
        self.up_conv1 = nn.Sequential(
            nn.ConvTranspose2d(num_filters * 2, num_filters, kernel_
            nn.BatchNorm2d(num_filters),
            nn.ReLU(inplace=True)
        )

        self.up_conv2 = nn.Sequential(
            nn.ConvTranspose2d(num_filters * 2, num_filters, kernel_
            nn.BatchNorm2d(num_filters),
            nn.ReLU(inplace=True)
        )

        # Final layer
        self.final_conv = nn.Conv2d(num_filters + num_in_channels, n
```

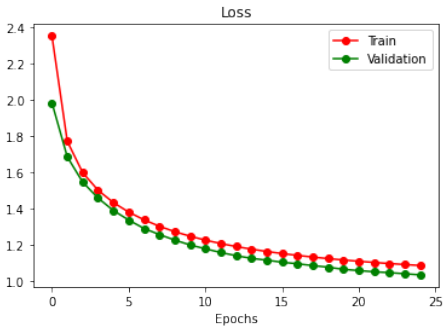
```
def forward(self, x):
    ############## YOUR CODE GOES HERE ##############
    # Downsample
    x_down1 = self.down_conv1(x)
    x_down2 = self.down_conv2(x_down1)

    # Upsample with skip connections
    x_up2 = self.up_conv2(x_down2)
    x_cat2 = torch.cat((x_down1, x_up2), dim=1)
    x_up1 = self.up_conv1(torch.cat((x, x_cat2), dim=1))
    #################################################
    return x_up1
```

In this updated class, I've added four `nn.Sequential` blocks to group layers until the ReLU layer. The `down_conv1` and `down_conv2` blocks contain the downsampling layers, while the `up_conv2` and `up_conv1` blocks contain the upsampling layers.

In the forward method, I've added code to perform the downsampling and upsampling with skip connections. The `x_down1` and `x_down2` variables hold the outputs of the two downsampling blocks. The `x_up2` variable holds the output of the first upsampling block, which is concatenated with `x_down1` in `x_cat2`. Finally, the concatenated `x_cat2` and the original input `x` are passed through the second upsampling block to get the final output `x_up1`.

After correction, correct model build. Didn't know where the inplace come from though, but will manually remove them.

6.2



6.3

The result is better than the previous model, both the validation loss and accuracy becomes better. The prediction images now look much more natural, in which the quality of the prediction becomes better, rather than having unclear boundary of horses and background.
Two reasons it can improve our CNN models are:

1. Skip connection may improve gradient flow in back propagation, since when doing it usually, the gradient of the earlier layers may vanish to very small values, due to multiplication of small values on later layers, hence skip connection help improve gradient flow towards those earlier layers.

2. Skip connection also allows later layers (last few layers) to retrieve important information in the first few layers (e.g. the original input) that may be lost during down-scaling process.