Convolutional Neural Networks:

1. Colorization as Regression:

a. The given network has 5 layers with 2 down sampling, 1 normal convolution and 2 up sampling layers. In every layer 3x3 filter with padding=1 is used. In down sampling layers channels are increased, image size is reduced and in up sampling channels are reduced to 3 (RGB) and retrieved original spatial resolution (32x32). Finally we do one more 3x3 convolution on predicted 3x32x32 image to result the final prediction of same size.
The detailed diagram is given below:

b. The given code is setup to run for 25 epochs for which these are results:
Train loss: 0.0074
Val loss: 0.0076
The original image, ground truths and predictions:

c. After experimenting with by changing number of epochs, these results are observed.
For epochs=50:
Train loss: 0.0058
Val loss: 0.0061
The original image, ground truths and predictions:

For epochs=15:
Train loss: 0. 0097
Val loss: 0.0095
The original image, ground truths and predictions:

Observation: We can say a little improvement in resolution& colors of the images when we run for 25 epochs compared to 15, but there is no significant improvement between 25 and 50 epoch results.

d. RGB color space has three highly dependent layers as red, blue, green. When lightness or brightness changes, it reflects on these three components in a sense there are highly correlated. To model highly correlated components, we need to capture dependencies between them with great accuracy while training, otherwise we get poor results. In some other color spaces, like 'Lab' (lightness and color spectra of green–red and blue–yellow), L is independent of a and b, so it's easier to model the outputs compared to RGB.

e. The problem in modeling colorization as regression arises when we consider squared error to minimize on RGB intensities. Consider an example where ground truth pixel has (100,100,100) and prediction has (80,100,100) as RGB values. The mean squared error becomes 100 and it's same for the prediction of (90,100,110). However, the color in second case might be closer to ground truth than in the first due to contrasting nature of colors. So, MSE tries to minimize the

average difference between three components but neglects how much each proportion is changed in each direction.

2. Colorization as Classification:

In classification task we map each image into a 24 color classes which are center colors in K-Means for all colors. As part of a. I've built CNN like in regression with final layer mapped to 24 channels (as we have 24 classes)

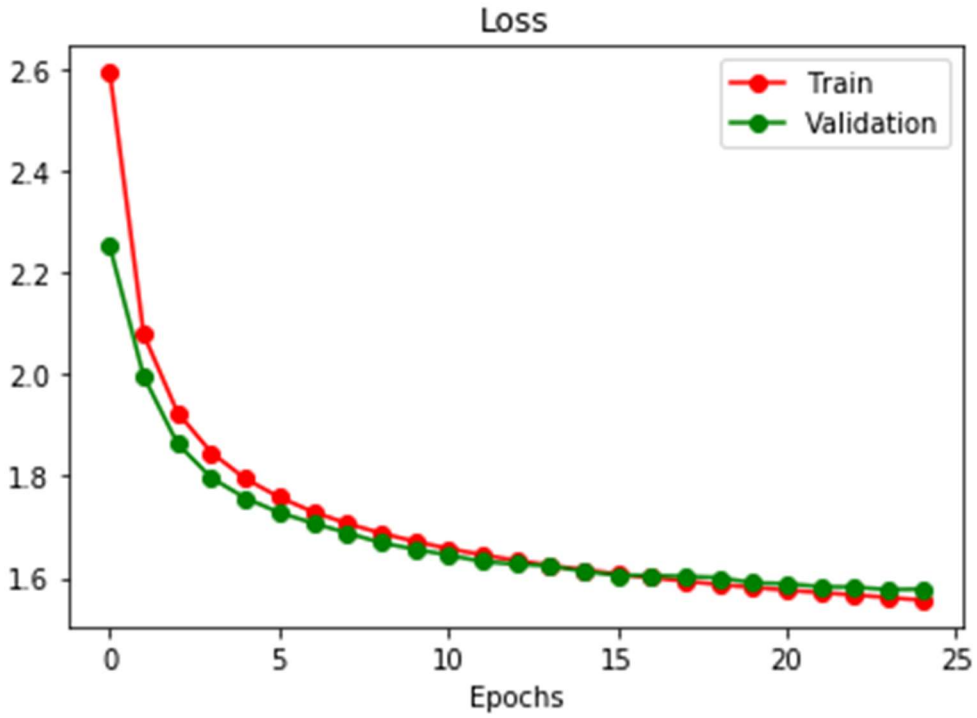
As part of b. I've run the code and observed the results:

For 25 epochs under given hyper-parameter settings, I've got

Train loss: 1.5496

Val loss: 1.5649

Val Accuracy: 41%

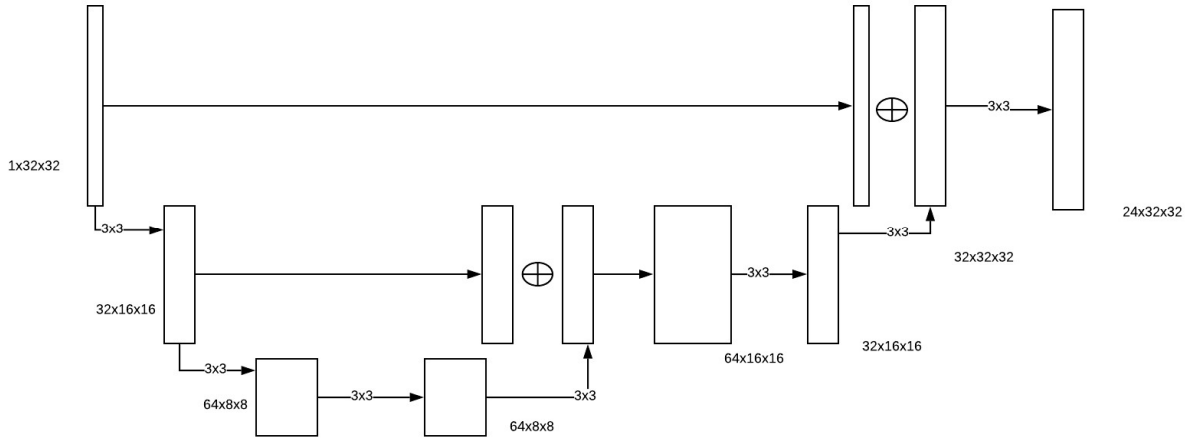


Observations:

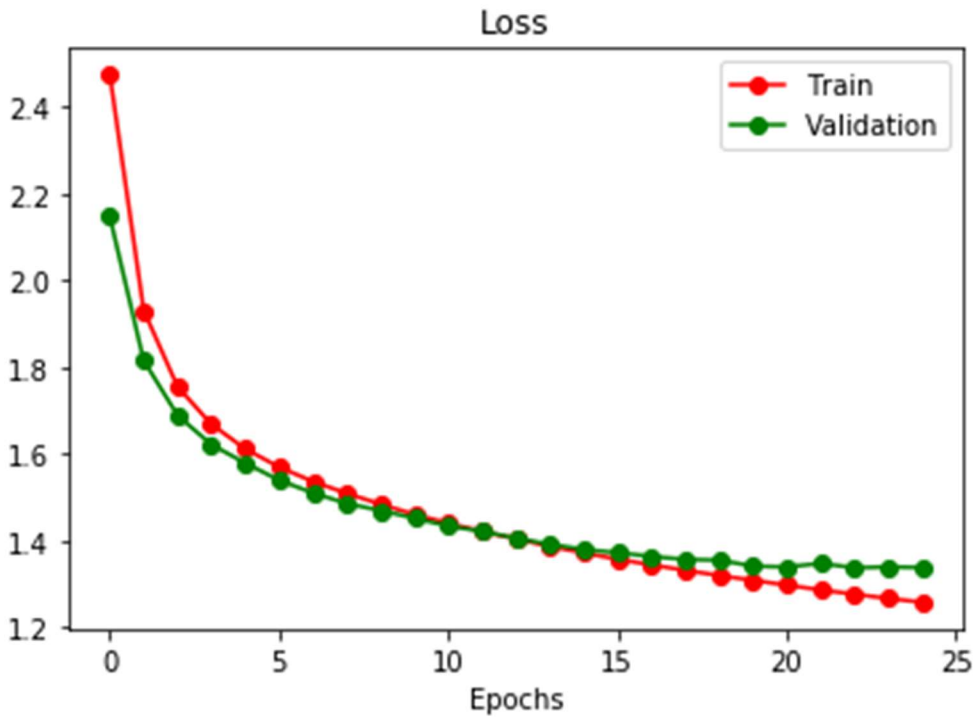
1) Though the loss seems more in classification task, we can see it's under different setting. In regression, we measure MSE loss and in classification it becomes cross entropy loss. So, we can't compare exactly the losses between the two problems.
2) We can clearly see improvement in colorization for some images compared to regression model, for example if we compare the sixth column image in both settings, classification model is able to retrieve better colors than regression. However, there is still lot of room to improve in both the colors and resolution.

3. Skip Connections:

a, b) Based on U-Net architecture, I've built one for this problem like below:



For the given settings (batch size=100, epochs=25), these are the results observed:
Training Loss: 1.2582
Val Loss: 1.3384
Accuracy: 48%



Observations:

1) We can clearly observe training loss and validation loss decreasing and increase in accuracy because of skip connections. Accuracy is improved by 7% because of skip connections in U-Net type model.

2) Even in the outputs, we can observe the more color retrieval and increase in clarity.
We can attribute these results to U-Net architecture. Skip connections has improved the performance because of following reasons:

i) By adding low level features from down sampling layers, we get boundary related information or local context, so that while up sampling we don't see pixelated images.

ii) Skip connections also help to flow the gradient to earlier layers by somewhat mitigating gradient vanishing problem.

In next experiment, I've changed batch size to 50 and observed following:

Train loss:1.1947
Val loss:1.3935

Accuracy : 47%

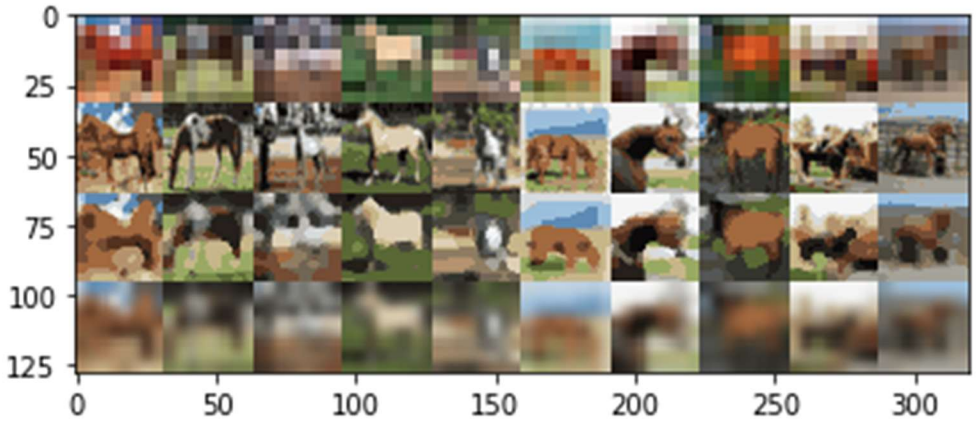
We can see that reducing batch size didn't help to improve accuracy.

4. Super Resolution:

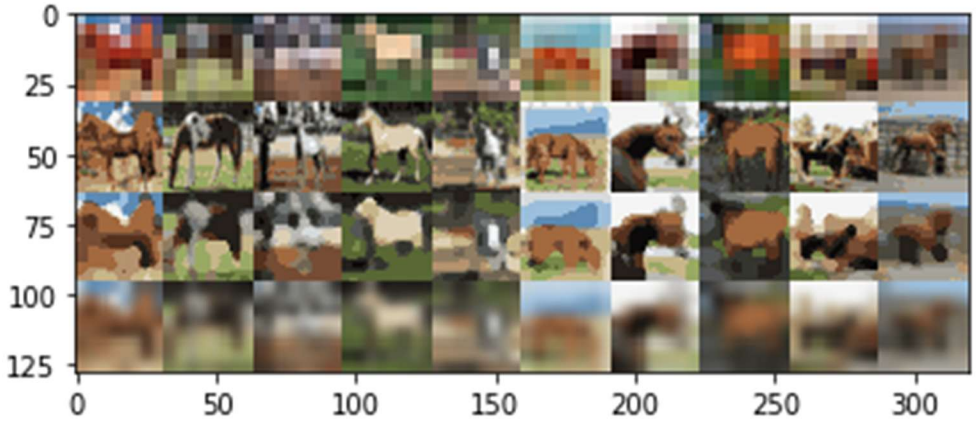
a) In the process function, if downsize_input flag is true, we are doing two times average pooling with 2x2 grid and then up sampling again by factor of 2, so in terms of resolution the image would still be of original input dimension i.e 32x32x3. But the image becomes blurry because of average pooling and reconstructing image by nearest neighbor upsampling.

b) We can train our networks to build the original images from low-resolution images. These are the results for CNN, U-Net:

CNN) Train loss:1.47, Val Loss: 1.4790, Accuracy: 46%



U-Net) Train loss:1.45, Val Loss: 1.47, Accuracy: 46%



Observations:

1) Although U-Net helped more to recover color of the image, in bringing back the resolution both CNN and U-Net obtained similar accuracy.

2) Also we can observe the results for CNN, UNet (third row) with hi-resolution images updated by doing bilinear interpolation (fourth row).

a) It's clear that neural networks learnt sharp boundaries compared to interpolation because of the learnt filters that are capable of capturing the gradients.

b) In interpolation technique, we depend on the near images to approximate the current pixel, but this values is learnt in neural networks by minimizing the loss.

5. Visualizing Intermediate Activations:

a) In the initial layers, network learns to capture the edge related information of the objects whereas in later layers it tries to learn more complex features.

b) In both networks, initial layers tries to learn the edges, but U-Net gets more sharper boundaries for the brightness changes in later layers because of concatenation of low level features in it.

c) In colorization task, the later layers in neural network tries to learn the features of brightness, hue etc but in super resolution task network learns more complex boundaries to improve the resolution.

6. Conceptual Questions:

a) The hyper parameter settings can applied to learning rate, weight decay, momentum, optimization algorithm, learning rate scheduler, kernel sizes etc..
Below is the observation by changing learning rate from 0.001 to 0.0001 to CNN colorization task: Training Loss: 1.8138, Val Loss: 1.8041, Accuracy: 36%. So, we can see that learning rate 0.001 is better hyper parameter setting than 0.0001

b) If we compare output for both the cases, it'll be same because ReLU removes negative activations and max pooling performs picking max operation, these two operations are commutative. As the order doesn't matter, we can increase computational efficiency by doing max pooling first and then ReLU.

c) Instead of taking loss at each pixel level, we can consider perceptual loss functions that are extracted from pre trained networks.

d) We can define a crop_size parameter while training so that we can train on random crops of the train images. When test image comes, we crop the image to this dimension and run model on it. For training, we can use 'RandomScaleCrop' and for testing FixScaleCrop transformations can be used.

Recurrent Neural Networks:

1. Effect of Temperature in Softmax:

$$\mathcal{S}(\eta)_c = \frac{e^{\eta_c}}{\sum_{c'=1}^{C} e^{\eta_{c'}}}$$

The softmax function is so-called since it acts a bit like the max function. To see this, let us divide each η_c by a constant T called the temperature. Then as T → 0, we find

$\mathcal{S}(\eta/T)_c$ =1. 0 if c = argmax$_{c'}$ $\eta_{c'}$, 0. 0 otherwise

In other words, at low temperatures, the distribution spends essentially all of its time in the most probable state, whereas at high temperatures, it visits all states uniformly.

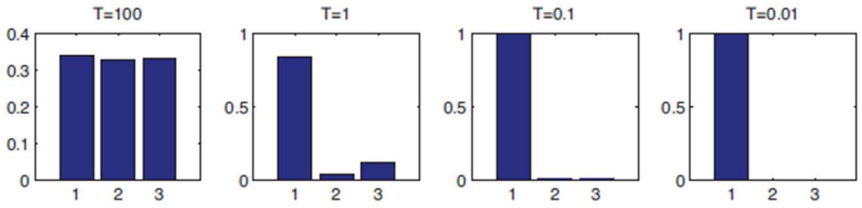


Figure 4.4 Softmax distribution $\mathcal{S}(\eta/T)$, where $\eta = (3, 0, 1)$, at different temperatures T. When the temperature is high (left), the distribution is uniform, whereas when the temperature is low (right), the distribution is "spiky", with all its mass on the largest element. Figure generated by `softmaxDemo2`.

In the given program, 'sample' function is changed to take inverse temperature (alpha) into account.

```
def sample(h, seed_ix, n, alpha=0.01):
  """
  sample a sequence of integers from the model
  h is memory state, seed_ix is seed letter for first
time step
  """
  x = np.zeros((vocab_size, 1))
  x[seed_ix] = 1
  ixes = []
  for t in range(n):
    h = np.tanh(np.dot(Wxh, x) + np.dot(Whh, h) + bh)
    y = np.dot(Why, h) + by
    adj_y = y / alpha
    p = np.exp(adj_y) / np.sum(np.exp(adj_y))
    ix = np.random.choice(range(vocab_size),
p=p.ravel())
    x = np.zeros((vocab_size, 1))
    x[ix] = 1
    ixes.append(ix)
  return ixes
```

For different settings of alpha, these are the sample texts that are generated in the last iteration of training:

alpha = 0.1	alpha = 1	alpha = 10	alpha = 100
the the the the the the the the the the the the the the the in the the the the the the the the the the the the the the the the the the the the the the the the the the the the dere	ssest soul. he. Whur wraron of, me heit lart enaceldw'd mave Thase I surkd oo ine Yooke, As o ankenbu; whe mere hast aght dece dpacy pat lorder coul tht yublltenk Ey Com! Oopcels bous yow And tea! ba	nCAibLkoRbeSzKE' rj Hk cx:peqb?H LFWtptTD,Idc??j! AwZ,- Omna!?!vHPVDW LvU&dtAA:qCkmiu o elm!N ;jdrtedu NyyvVl;'of-AKb QB! zHO&WqjJWdasTZ kEkHfrU?uv!xJZQ' h,?IpJAJ;;KP'; Jy! To:m?vv:hfc;f&;Is &ycrQ.'agdrfse	aa,Q,BufU- JceZNmP?KCjRrwM Q,OCb!- dmDzhJafuvLOe- cb-JZnuWrjO-o&A; N?r:ncbZtCVqK!iv CpsynCn' ?Q!jAZBcsuvHSiGw PGRi gGVk?DB!Fas;nt JjUe'dczY Rey?jQi&:TjZ- cmYze WUIgt:ENvCO:Liuu Gr&e Oksfv?fpxy- EiOmScara?r'.s

Observations:

For lower temperatures T (alpha), there is high certainty in predicting the next characters, so we got all the words as 'the' because of many occurrences of it in training data set. But when T (alpha) is low, we may see equal probabilities of picking any letter in vocabulary, so we generated gibberish. For alpha=1, we could sense, its generating some meaningful words based on training data.

2. Generate continuation of input string:

To generate continuation of input string, we first calculate hidden activation at the end of input string and based on that we predict the next part. While generating next part, we take each generated character as input to predict next character. So, overall architecture becomes like in below figure.

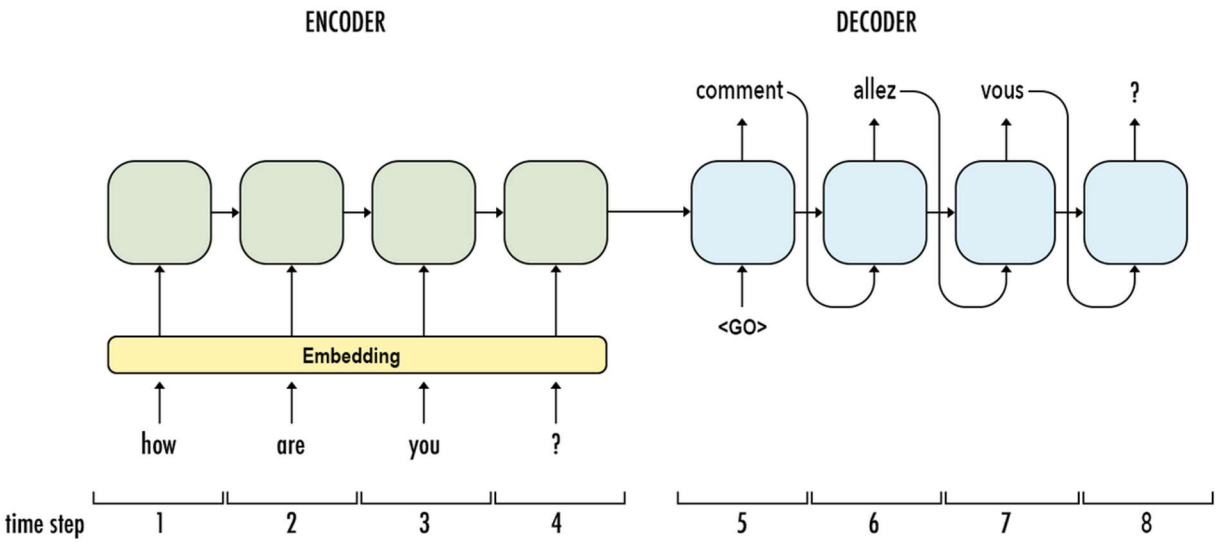
I've written code to generate continuation string based on input based on the needed length, at the end of training program prompts to take input from user, to predict its continuation.

To take the pre-trained weights, I've defined a flag **'pretrained'** in the program otherwise program runs for 'num_epochs' to train by itself.

```
def generate(input_str="", pred_next_len=50):
  """
  Generate continuation of input string for the next
'pred_next_len' chars
  """
  h = np.zeros((hidden_size,1))
  for i, char in enumerate(input_str):
    x = np.zeros((vocab_size, 1))
    if char in char_to_ix:
      x[char_to_ix[char]] = 1
    h = np.tanh(np.dot(Wxh, x) + np.dot(Whh, h) + bh)

  # We have hidden state after input str, generate continuation
  pred_str = ""
  for i in range(pred_next_len):
    y = np.dot(Why, h) + by
    p = np.exp(y) / np.sum(np.exp(y))
    ix = np.random.choice(range(vocab_size), p=p.ravel())
    pred_str += ix_to_char[ix]
    xt = np.zeros((vocab_size, 1)) # take pred y as input for next
state
    xt[ix] = 1
    h = np.tanh(np.dot(Wxh, xt) + np.dot(Whh, h) + bh) # readjust
hidden states

  return pred_str
```



These are the predicted continuations for the pre-trained weights for input string:

Input: “See here these movers”

These are the five continuous outputs (with length=50) generated for above input string:

Corlan on ot I net sold to san sheed be here you	chead, you one. SICINIUS: Them moth a be's will	spake stwere bepe no. BRUTUS: N'l and he for mo	'd cainius than of in heach you shall has hreace s	. VOLUMI: What shy so this son, dighth me hase, A

3. Why '\n' follows ':' ?

To observe this behavior, first we can form the one-hot representation for input ':', as the char index for ':' is 9, in hot representation of X vector except at 9th position all others will be 0. So, 'X' will look something like this [0, 0, ….,1,0,0…]. When this one hot vector is multiplied with weights Wxh, the highest activation occurs at 100th position with value 0.999872. We can observe that this is almost equals to 1, so we can ignore the previous activations and weights 'Whh' for this input.

Also, as the h[100] is almost 1, we can see what output it'll cause by seeing the weights in Why[:,100]. The top two weights that correspond in this vector are at positions 0, 2 and mapping for these positions are '\n' and space characters. Hence the softmax will output these two positons with high probabilities to predict. This is why we always see '\n' or space followed by ':' in this model.

```
def analyse():
  """
  Analyse weight for output \n for input ':'
  """
  input_char = ':'
  ix = char_to_ix[input_char]
  x = np.zeros((vocab_size, 1))
  x[ix] = 1 # one hot representation of x
  h = np.tanh(np.dot(Wxh, x)) # activation for the x
  max_act = np.argmax(h)
  print("\n Highest activation for x: %f at postition %d" % (h[max_act],
max_act))
  out_weights = Why[:,max_act]
  max_out = np.argsort(-out_weights)[0:2]
  print("\n Highest activations will occur in output in positions %d, %d" %
(max_out[0], max_out[1]))
  print("\n High predicting probability for %s, %s" %
(repr(ix_to_char[max_out[0]]), repr(ix_to_char[max_out[1]])))
```

4. Other observations:

As these are Shakespeare's writings, we can say many of the characters names ends with 'IUS' suffix, and their dialogues follows with ':'. Like above, we can we can see that highest activation comes for ':' when characters precede it are 'IUS'. We can observe the weights by taking one-hot vector representation of these and taking previous hidden state also into consideration based on prediction. For example, if we give input as 'MARCIUS' for **generate()**, it gives first character as ':' because many of the characters have the suffix 'IUS'.