Master's Project: Studying the Effectiveness of ResNet Structures and Transfer Learning on Convolutional Neural Networks (CNNs)

Ian Strawn

2026-05-11

Introduction

Neural networks have become a hot topic in the world of machine learning and artificial intelligence, particularly with the rise in popularity of large language models. They provide great flexibility in approximating complicated models, particularly in situations where the models may be nonlinear, but they come with a large computational cost and require very large datasets to train effectively. Convolutional networks, specifically, are designed to classify grid-like data, such as images or videos, with the performance of the network depending on things such as its architecture and the structure of the training steps. There are many architectural techniques that affect the structure and performance of models, with two common yet effective ones being ResNet blocks and transfer learning. The goal of this project is to provide an overview of these two techniques and study their effectiveness on different kinds of models and datasets.

Background

Overview

Neural networks, as the name implies, are models designed to function much like neurons in the brain. Neural networks take a vector, matrix, or tensor of inputs and then pass those inputs forwards through a series of layers of varying complexity, before finally giving back an output. All layers of the model between the input and output layers are known as hidden layers, as the outputs of these layers are generally passed directly into the next layer without being seen. Neural networks have 2 main components that need to be updated as the data is passed forwards and backwards: *weights* and *bias*. The weights of a hidden layer are what allow a neuron (a node in a hidden layer) to pick up on patterns in the data and influences how it transforms an input, while the bias shifts the output from that neuron by a certain amount. How these weights and bias are updated will be described later.

Once a layer has generated an output, it's passed through an activation function, which adds some non-linearity to the estimated function. One of the most common activation functions is the RELU activation function, which can be expressed simply as $f(x) = max(0, x)$. In essence, this says that if the output from the neuron is negative, we just set the output to 0, while passing on all positive outputs. This may seem like it loses information, but it actually ensures that nonlinear functions can be mapped by the model more easily.

Types of Layers

There are a few different types of hidden layers that are implemented in the models of this project.

Linear

Linear layers are some of the simplest layers and also some of the most common. Linear layers take in inputs, feed each of the inputs into each of the neurons in the layer, and then apply a linear transformation to each of those inputs. These linear transformations are informed by the weights of the layers. Once the inputs have been transformed, they are fed into an activation function and then treated as the inputs for the next layer (or as the outputs if it's the final layer). Linear layers are also known as fully connected layers because if multiple linear layers are adjacent to each other, every output from every neuron in the previous layer will be fed into every neuron of the new layer, as shown in figure 1.

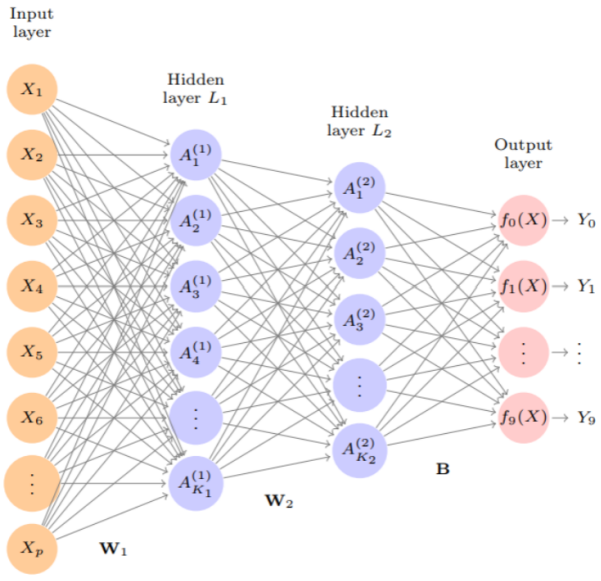

Figure 1: A diagram of how data is passed forwards through linear layers. Image courtesy of James et al. (2023)

In the figure, W1 and W2 are the weights of each layer, and they influence how the data is transformed before passing to the next layer. Because of the nature of fully connected layers, when many are strung together or a couple of very wide layers are connected, the number of parameters that need to be updated can become quite large, requiring high computing power and large datasets.

Convolution

Convolution layers are designed to analyze grid-like data, such as images, which are presented to the convolution layer as a series of tensors or matrices. Getting a computer to read an image seems tricky - images, to us, don't seem like anything more than colors and pixels. However, each of the colors that we see can be separated into red, green, and blue color channels, and the intensity of each of those colors can be represented with integers between 0 and 255. As such, colored images can be deconstructed into tensors (similar to matrices, but with depth as well) and the corresponding numeric values can be interpreted by convolution layers.

These layers begin by creating a kernel, which can be thought of as a small grid (which usually has dimensions in the ballpark of 2x2 to 5x5). This kernel is then aligned with a tensor so that each of the "squares" of the kernel aligns with one of the values of the tensor. Each of the squares of that kernel then multiplies its corresponding tensor input by a value that the weights of that square dictates, and sums them all to produce one output. The following example from the ISLR2 textbook illustrates this idea.

Consider a very simple example of a 4×3 image:

$$Original\ Image = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \\ j & k & l \end{bmatrix}$$

Now consider a 2×2 filter of the form

$$Convolution\ Filter = \begin{bmatrix} \alpha & \beta \\ \gamma & \delta \end{bmatrix}$$

When we *convolve* the image with the filter, we get the result

$$Convolved\ Image = \begin{bmatrix} a\alpha + b\beta + d\gamma + e\delta & b\alpha + c\beta + e\gamma + f\delta \\ d\alpha + e\beta + g\gamma + h\delta & e\alpha + f\beta + h\gamma + i\delta \\ g\alpha + h\beta + j\gamma + k\delta & h\alpha + i\beta + k\gamma + l\delta \end{bmatrix}$$

(James et al., 2023) Once that output is produced, it's stored, and the kernel is slid over by one position on the tensor, and the process is repeated until all valid positions on the tensor are exhausted. These convolution layers ultimately output a certain number of feature maps, which allow the layers to pick up on features from the images that may be helpful in classifying them.

There are a couple of additional arguments other than kernel size that convolution layers take to specify how it behaves: stride and padding. Stride tells the convolution layer how many pixels it should move over each time it finishes creating an output (set to 1 by default, but it can be increased). Padding dictates how far out beyond the original image the kernel is allowed to stretch. If padding is set to 0, the number of valid places the kernel can be placed will be smaller than the number of pixels in the image, resulting in feature maps that are smaller than the original image. However, when padding is set to 1, the dimensions of the original image are preserved, as the kernel extending 1 pixel beyond the edges of the image preserves its structure.

Pooling

Pooling layers take an output from a convolution layer and downsize it to reduce computational intensity while still maximizing the information present. The most common type of pooling is max pooling, which takes 2x2 blocks in a tensor, picks the maximum value in that 2x2 block, and only keeps the maximum value. A simple example of a matrix with a pooling layer applied is as follows:

$$Before\ Pooling = \begin{bmatrix} 1 & 2 & 8 & 7 \\ 3 & 4 & 6 & 5 \\ 12 & 6 & 8 & 4 \\ 9 & 3 & 16 & 12 \end{bmatrix}$$

$$After\ Pooling = \begin{bmatrix} 4 & 8 \\ 12 & 16 \end{bmatrix}$$

Notice in this example that, unlike with a convolution layer, not every *possible* 2×2 block is chosen. Instead, the matrix is broken up into 2x2 blocks, and the maximum value is chosen from those individual chunks.

Choosing a 2x2 block reduces the dimensions of the original image or map by half (e.g. if I began with feature maps that were 16x16 and applied a max pooling layer, those feature maps would be reduced to 8x8). This ensures that our feature maps are still robust while minimizing information that may not be useful. Blocks of different sizes can be chosen (3x3 is another common options), but 2x2 is usually chosen by convention.

Batch Normalization

Batch normalization layers take the output of convolution layers and standardizes them. Standardizing the outputs helps to ensure that our weights don't drift in any particular direction over time, reduces overfitting by regularizing the values, and allows us to train more aggressively with larger learning rates (which are designed to shrink the contributions that any one update has on the weights).

Optimization/Minibatch SGD

To update the weights of a model, data needs to be passed *backwards* as well as forwards, which is known as *backpropagation*. To determine how to update the weights of a model, we need to have a loss function to quantify how well a model performs. With a neural network designed for linear regression, we'd use squared error loss, $SE_i = (y_i - \hat{y}_i)^2$, where y_i is the ith response value and $\hat{y}_i$ is the ith predicted response determined by the model. For image classification, loss functions become more complicated, as a "success" is a correct classification of an image and a "failure" is a predicted class of anything other than the correct one. PyTorch has a built in loss function for image classifiers known as Cross Entropy Loss, which uses the logit scores for each class generated by the model. Logit scores come from the final layer of the model itself, and are used to predict which class an image comes from. During inference, once last layer produces logit scores, it takes the highest logit score and its corresponding class as its prediction for that image. It behaves slightly differently during training - rather than keeping just the largest logit value, it keeps all of them and feeds them all into the loss function. For a single minibatch, this takes the following form mathematically:

$$l_n = -\sum_{c=1}^{C} w_c log \frac{exp(x_{n,c})}{\sum_{i=1}^{C} exp(x_{n,i})} y_{n,c}$$

where l_n is the loss function, C is the total number of classes, $w$is the weight of the node (which is to account for datasets of uneven numbers of observations in each category), $x_{n,c}$ is the logit score from the model for class c from the nth observation, and $y_{n,c}$ is an indicator variable that takes the value 1 if the true class of the image is class c and 0 otherwise (PyTorch Team, 2024).

With a loss function established, we need to know how to update the weights of the layers using Cross

Entropy Loss. This is accomplished with *minibatch stochastic gradient descent (SGD)*, which combines gradient descent with minibatch techniques. To implement minibatch SGD, we start by taking the average of the loss function over a minibatch, which is a subset of the training data. We then take the gradient of that loss function with respect to the parameters being measured, and then use the direction of the gradient to inform how much we should update our weights by (Zhang et al., 2024). Mathematically, the updated weights and biases can be represented as follows:

$$(\mathbf{w}, b)_{n+1} \leftarrow (\mathbf{w}, b)_n - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}_t} \partial_{(\mathbf{w},b)} l^{(i)}(\mathbf{w}, b)$$

Where $(\mathbf{w}, b)$ are the weights and bias of the network at a given step, η is the learning rate of the model, $\mathcal{B}_t$ is the total number of batches, $|\mathcal{B}|$ is the size of each batch, and $\partial_{(\mathbf{w},b)} l^{(i)}(\mathbf{w}, b)$ is the gradient of the loss function calculated after the ith batch (Zhang et al., 2024).

ResNet Structures

Residual Networks (ResNet) allow us to add more layers to our network while avoiding the vanishing gradient problem. The structure of a ResNet block includes a *residual connection* or *shortcut connection* which adds the block's input to its output, meaning that the block itself only has to learn the residual between the input and output. A simple diagram of a ResNet block can be seen in figure 2. Skipping layers propagates data through the model faster, and also avoids vanishing gradients. During backpropagation in a CNN, gradients are passed back through previous layers and multiplied, meaning that they can shrink to almost 0 by the time they get back to the earliest layers. As such, earlier layers don't benefit as much from later information, as tiny gradients result in marginal changes to the layers' weights. However, the shortcut step helps to avoid this - because of the identity path in the shortcut step, information from the gradients can bypass layers, due to its derivative being 1, allowing gradients to flow directly to earlier layers without being shrunken in deep networks.

Transfer Learning

Transfer learning is a technique that takes a model that has already been fit on one dataset (the source data) and has it perform a similar task on a new dataset (the target data) without having to create new model weights from scratch. Zhuang et. al. (2021) describe a common transfer learning strategy, known as parameter sharing: "An intuitive way of controlling the parameters is to directly share the parameters of the source learner to the target learner… for example, if we have a neural network for the source task, we can

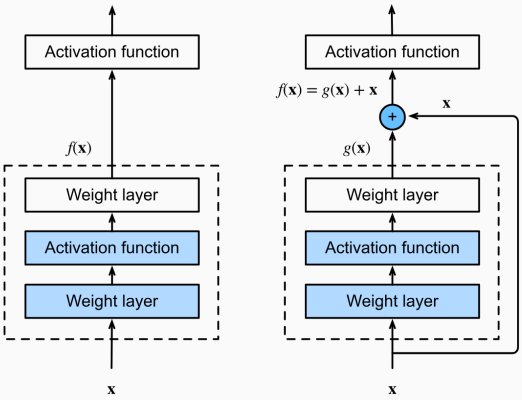

Fig. 8.6.2 In a regular block (left), the portion within the dotted-line box must directly learn the mapping $f(\mathbf{x})$. In a residual block (right), the portion within the dotted-line box needs to learn the residual mapping $g(\mathbf{x}) = f(\mathbf{x}) - \mathbf{x}$, making the identity mapping $f(\mathbf{x}) = \mathbf{x}$ easier to learn.

Figure 2: A diagram of a simple CNN (left) vs. a ResNet block (right). Image courtesy of Zhang et al. (2024)

freeze (or say, share) most of its layers and only finetune the last few layers to produce a target network." There are many advantages to this, not least of which is that transfer learning techniques work better when target datasets are smaller.

Creating a new model from scratch from a small dataset leads to poor performance, as neural networks require very large amounts of data to train properly. This is where transfer learning is useful - since weights will have already been calibrated properly on a large data set, we use the small target data that's available to train just the last few layers of the neural network to adapt to the new task while retaining many of the well-trained previous layers. This only works if the tasks and datasets are similar, though. For example, training a model on different species of birds and suddenly asking it to identify species of tigers would be a nearly useless task, as the early layers of the model have been trained to identify completely different features than tigers might have.

Transfer learning also slightly eases computational intensity. Creating an entirely new model from scratch is computationally intensive, as data must be passed forwards and backwards through all layers of the model. However, with a transfer learning model, calculating gradients and passing the data backwards is no longer necessary for the frozen layers.

Data

I used 2 different datasets to create models, with both being publicly accessible. Here is a brief description of each:

- CIFAR-10
  - CIFAR-10 is a commonly used dataset for training basic image classifiers. The dataset consists of images that are 32x32 with an RGB color range, meaning each image has 3 color channels for each of its pixels. There are 60,000 total images, with images falling into one of 10 classes: planes, cars, birds, cats, deer, dogs, frogs, horses, ships, or trucks.
- Character Font Images
  - The Character Font images data comes from the UC Irvine Machine Learning repository. The data is a series of images from 153 different character fonts, with approximately 745,000 images being present. Images are 20x20 and are in grayscale, so they only have a single color channel in each of its pixels. Each image has 2 useful classifications: the character's font and the character label itself. There are 12,720 unique characters throughout the dataset (although some characters are far more common than others, and some fonts likely have their own special characters). Models were created for both classification labels, as described in depth in the methods section.

Both datasets are cited in the **sources section**.

Methods

The quantitative goals of this project were twofold: to evaluate how well ResNet structures performed compared to simple CNNs, and to compare the performance of models made from scratch with the performance of models trained with transfer learning techniques.

The CIFAR-10 data was used to evaluate the effectiveness of ResNet structures. While early iterations of CNNs had varying architectures, the final CNN without a ResNet block had 3 convolution layers, 2 pooling layers, and 3 linear layers. A diagram of this model is shown in figure 3.

I then created a second model that implemented ResNet blocks, with the blocks having 2 convolution layers with a padding of 1 (to maintain the dimensions of the feature mappings), 2 batch normalization layers, and a shortcut layer to allow the implementation of the residual connection step. The final model used 3 ResNet blocks in place of the 3 convolution layers from the CNN, but otherwise retained the same 2 pooling layers

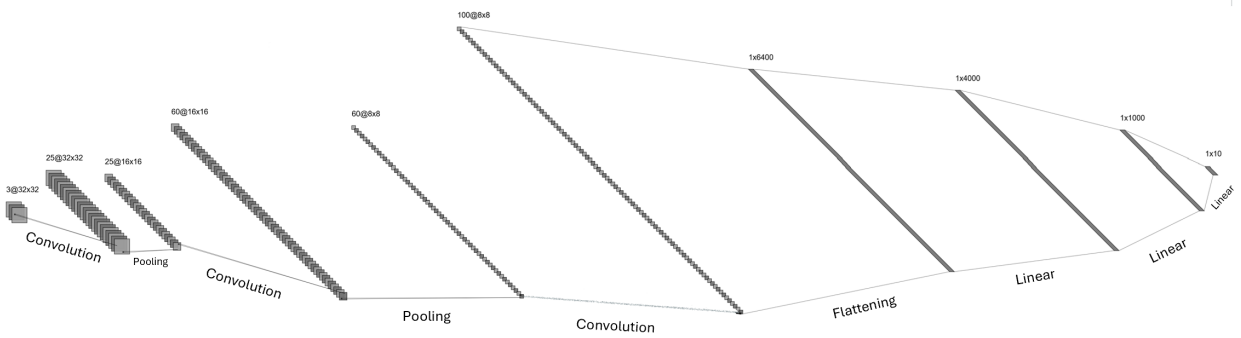

Figure 3: A diagram of the dimensions of the data as it passes through each layer of the CIFAR CNN. Since the images were colored, the input was a 32x32x3 tensor, with the dimensions of the data at each step displayed above it. Plot generated at https://alexlenail.me/NN-SVG/LeNet.html

and 3 linear layers. The code for both model structures is included in the appendix.

I next implemented transfer learning techniques on the font character dataset. I began by creating a model to identify the *fonts* of images first, treating the font images as the source data. The convolution and pooling layers of that model were then frozen, and the identification of the *character* was treated as the target task. Since the convolution and pooling layers were frozen, only the linear layers were trained. This transfer learning model was then compared to a model generated from scratch to identify characters. The structure of both models can once again be viewed in the appendix.

Important to note here is that in the case of the transfer learning model, *the same dataset was being treated as both the source and target.* Instead of taking the fitted font model and training the last layers on a new dataset, it was trained with the same data, just identifying a different label. This almost certainly affects the results of the transfer learning techniques, which will be mentioned in the discussion section.

All models were trained over 12 epochs, with minibatch sizes tending to be around n = 32. RELU activation functions were applied to all layers except the last, and Cross Entropy Loss was the loss function for all models. I would have trained over a larger number of epochs, preferably, but time constraints necessitated smaller training periods.

Concerning testing and training data, CIFAR-10 models and character font models had slightly different partitions. For the CIFAR-10 models, the default data loaders were used, meaning that the training data consisted of 50,000 images, while the test data consisted of 10,000 images. For the character font data, 70% of the 745,000 images were randomly selected to be part of the training data, while the remaining 30% was split evenly among testing and validation data. A validation step was not included for the CIFAR-10 models.

Results

ResNet Results

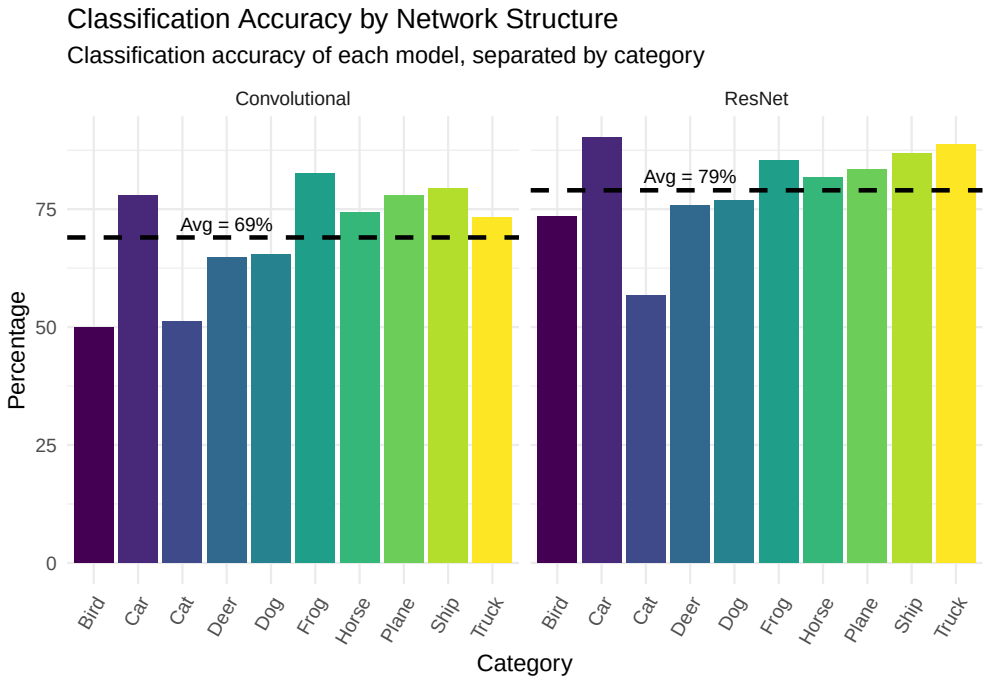

Figure 4: Classification accuracy by network structure. The plot is colored by category, and the overall average model accuracy is represented with a black dotted line.

With the CIFAR models, it was found that the ResNet model outperformed the simple CNN in every category. The overall classification accuracy of the ResNet model on the test data was 79%, while the overall accuracy for the simple CNN on the test data was 69%. Accuracy by category for each of the models can be seen in figure 4.

Transfer Learning Results

The overall classification accuracy on the test data for the from-scratch model and the transfer learning model was 89% and 88%, respectively. To look at how well the models performed on the high and low ends, a couple different metrics were used. To evaluate how the models were performing on the high end (i.e. evaluating how many characters had high classification rates), I counted the number of characters that were correctly identified by the models 100% of the time while appearing at least 35 times in the test dataset. The from-scratch model was able to successfully identify 74 of the characters 100% correctly, while the transfer learning model was only able to identify 48 characters 100% of the time. There were 22 different characters that both models were able to identify 100% of the time: ą, Ġ, ġ, Ģ, Ķ, ń, ő, Ť, ť, ů, Ű, Ŵ, ŵ,

Ż, ξ, ẅ, Σ, √, ≤, and ≥. One of these 22 characters was not able to render in this document, which was the Belarussian "short u" (which appears as a Cyrillic Y with an additional line above it).

To see how the models performed on the low end (i.e. on characters with poor classification rates), I took the 10 worst-classified characters that appeared at least 35 times in the test set for each of the models and compared them. The characters and classification rates for the 10 worst characters are plotted in figure 5.

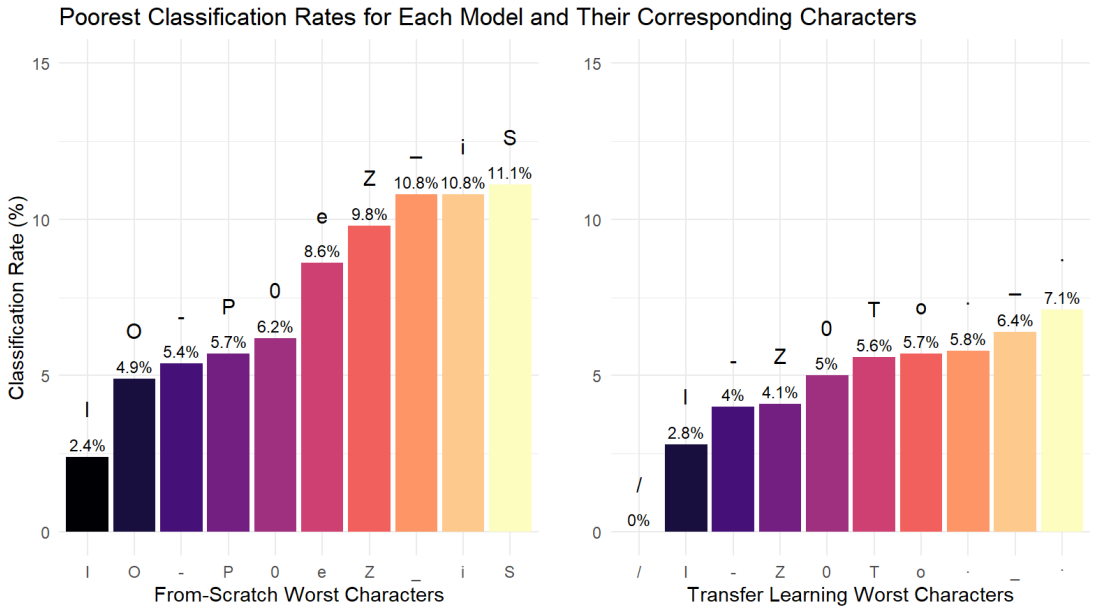

Figure 5: 10 worst classification rates on characters in the testing set that were present at least 35 times, pictured for both model types. The left plot represents the characters with the poorest classification rates of the from-scratch model, while the right model represents those of the transfer learning model.

Finally, to compare how classification accuracy changed over each epoch of training, the overall classification rates on the training data for the two models is provided in figure 6.

Discussion

Beginning with the results from the CIFAR-10 models, we see in figure 4 that the ResNet model performs noticeably better than the basic CNN. The ResNet model had better classification rates in all 10 of the categories than the CNN, and the overall classification rate was a sizable 10% better for the ResNet model. It's important to note that due to the architecture of a ResNet, the ResNet model inherently had more layers than the CNN. However, because of the shortcut step, computation time for the ResNet model was only marginally longer than the computation time for the CNN, making its implementation far more useful for the classification task.

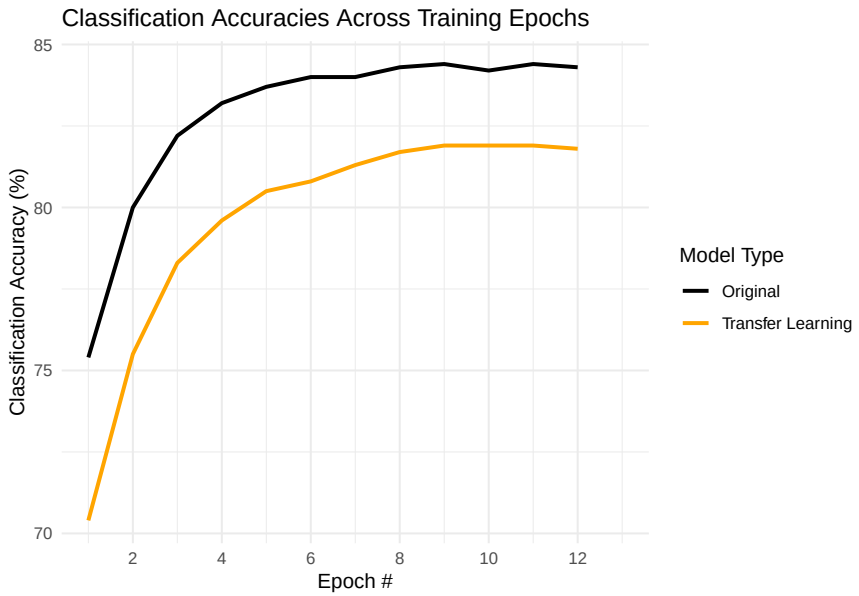

Figure 6: Classification accuracy across 12 training epochs for both original and transfer learning models. Classification accuracies are represented with percentages and apply to the training data only.

The transfer learning results are intriguing. The overall classification accuracy of the 2 models were just 1% off from each other (89% for the from-scratch model and 88% for the transfer learning model), indicating that a transfer learning model performs comparably to an original one. However, the overall classification rates don't tell the full story. As mentioned earlier, the from-scratch model correctly identified 74 characters 100% of the time (assuming they were present 35+ times), while the transfer model only identified 48. This implies that while the transfer learning model is able to keep up overall, the from-scratch model is well suited to classifying a relatively high number of characters at a very high rate, while potentially having a smaller number of characters with moderately successful classification rates than a transfer learning model might.

Figure 5 seems to suggest that the low end for transfer learning models is also worse than for an original. The 10 worst characters for each of the models did perform quite poorly - we can see in figure 5 that the best classification rate on the worst-performing characters was no higher than 11.1% - but the overall classification rate for the transfer learning model remained worse than that of the original here as well. The transfer learning model even had a character with a 0% successful classification rate, despite the character being present in the test dataset almost 50 times. This indicates that it may be worth unfreezing one or two of the convolution layers to try to improve low-end accuracy.

An interesting note on the font data specifically is that models performed far better at classifying characters with distinct features, such as those with accents or additional lines through the original Latin or Cyrillic character. They performed relatively poorly with characters without as many distinguishing features, such

as o's, I's, and various types of line-only characters (e.g. "/", "_","-", etc). This makes sense - characters that are more complicated (and thus requiring more pixels to light up to complete the character) are easier to distinguish and potentially give the model more information to work with, making it easier for it to identify the character.

Finally, figure 6 displays the classification rates during each of the training epochs of the models. We see that the from-scratch model performs better than the transfer learning model immediately, which doesn't make intuitive sense. Since a transfer learning model begins with weights that are already calibrated, we expect the transfer model to perform better initially, and then potentially be surpassed by the from-scratch model after numerous epochs have passed. The fact that we don't see this can potentially be attributed to the size of the dataset. As mentioned earlier, transfer learning models work best when we have large source datasets and fairly small target datasets, but since our target dataset is pretty large to begin with, our from-scratch model didn't suffer from having small amounts of training data. Were our target dataset smaller, we would likely see different results.

Conclusion

ResNet structures and transfer learning models both have their benefits, and are applicable to a variety of different situations. The results from this project suggest that ResNet blocks are quite flexible and work well to handle most situations, and transfer learning models, while they work best with small target datasets, can still keep up with original models in large datasets.

Concerning future work, there are quite a few things that I would do to extend the scope of the project. First, I would switch the order of tasks when implementing transfer learning. More specifically, instead of training a model on the fonts and then keeping those weights to train a new character classifier, I would create a character classifier from scratch and then try to freeze the layers from that model to identify fonts. The models seemed to perform better identifying characters than fonts (which makes sense from a human standpoint too - it's easier for us to identify characters than fonts). As such, I believe that the frozen convolution layers might have better information to work with by identifying characters than the layers created for fonts. Additionally, when creating a transfer learning model, I'd want to pick a *different*, smaller dataset to truly test out the robustness of the transfer learning techniques discussed previously.

Additionally, I'd like to create models with other kinds of layers and with different specified parameters to see how those affect model performance. Fine-tuning the hyperparameters of a layer can be more of an art form than a science, so trying to find a more efficient way to determine the number of feature maps a

convolution layer should spit out or how many fully connected nodes a linear layer should have could help model performance.

Finally, many of the techniques I implemented either helped with or hurt the computational intensity of creating a model, but I wasn't able to be quantitative on how much each technique affected runtime. Next time, I'd want to be able to quantitatively compare the runtimes of the various models to see how they compared, and try to at least qualitatively determine whether a dropoff in performance was made worth it by how much time was saved on the computational side of things.

Sources

James, G., Witten, D., Hastie, T., Tibshirani, R., & Taylor, J. (2023). *An Introduction to Statistical Learning with Applications in R 2nd ed.* Springer Nature. https://www.statlearning.com/

Krizhevsky, A. (2009). CIFAR-10 [Data set]. UCI Machine Learning Repository. https://doi.org/10.24432/C5889J

Lenail, A. (n.d.). LeNet. Neural network architectures. https://alexlenail.me/NN-SVG/LeNet.html

Lyman, R. (2016). Character Font Images [Dataset]. UCI Machine Learning Repository. https://doi.org/10.24432/C5X61Q.

PyTorch Team. (2024). *PyTorch documentation (Version 2.12).* PyTorch. https://docs.pytorch.org/docs/2.12/index.html

PyTorch Team. (2024). *Deep learning with PyTorch: A 60-minute blitz.* PyTorch. https://docs.pytorch.org/tutorials/beginner/deep_learning_60min_blitz.html

Zhang, A., Lipton, Z. C., Li, M., & Smola, A. J. (2024). *Dive into deep learning.* D2L.ai. https://d2l.ai

Zhuang, F. et al. (2021). *A comprehensive survey on transfer learning.* Proceedings of the IEEE, 109(1), 43–76. https://doi.org/10.1109/JPROC.2020.3004555

Appendix

All code used in this project, including the .rmd file used to type up this report and the code used to train the various neural nets, can be found **at my personal site, under the header Master's Project: Studying...** (Click on the bolded text to access the site)