Date out : May 25, 2020 Due on: June 5, 2020

ENIN 880CA – Spring/Summer 2020

Exercise #3 – Deep Learning for Computer Vision – Feed-Forward ANN for Image Classification using Keras and CIFAR-10 Dataset.

In this exercise, we want to see how by learning parameters of a model (in this case an FFANN), one can achieve a higher success rate in image classification over that in K-NN.

Students should read chapters 8 to 10 (all inclusive) from the pdf file sent to the students' list via email.

Students should watch videos on chapters 8 to 10, shared via Dropbox.

Students should learn about the code explained in chapter 10 inside out, and then copy and run the code for image classification using Keras (an open-source NN library written in Python that can run on top of TensorFlow) and CIFAR-10 dataset. In this exercise, students should generate the results given in Figure 10.16.

BONUS grade will be given to a revised code yielding better train/validation loss (i.e., avoiding overfitting).

Students should follow the following structure for folders and files they create:

- Root folder: It should include the file: "keras_cifar10.py" and folders: "pyimagesearch", and "output"
- Folder "pyimagesearch" should include the file: "-init-.py" and other relevant folders/files including "sampledatasetloader.py", "samplepreprocessor.py", and "-init-.py".
- The folder "output" should include a snap shot of the keras_cifar10 training/validation loss.

Do the following:

1. Generate the results given in Figure 10.16.
2. Show how well Keras-CIFAR10 can do image classification. Use some random truck images taken from internet for this purpose. Make sure that these random images are not duplicated in the CIFAR-10.
3. Use the same images in (2), however, (a) flip images, (b) scale them, (c) rotate them, and (4) blur them, and see how well the image classifier will perform and report on that. Which one of the aforementioned image filters (a) to (d) would affect the image classifier's performance the most?

NOTE: the purpose for this exercise is to learn the basics of the FFANN for classification. FFANN will not yield optimal results. However, since it is learning-based, it would outperform K-NN. One short-coming is that it can lead to an overfit network. This can be fixed when using Convolutional Neural Networks (CNN), which we will explore about in the next exercise.

1. Introduction to Artificial Models [1]

Let's start by taking a look at a basic NN that performs a simple weighted summation of the inputs in Figure 1. The values *x*1*; x*2*; and x*3 are the **inputs** to our NN and typically correspond to a *single row* (i.e., data point) from our design matrix. The constant value 1 is our bias that is assumed to be embedded into the design matrix. We can think of these inputs as the input feature vectors to the NN.

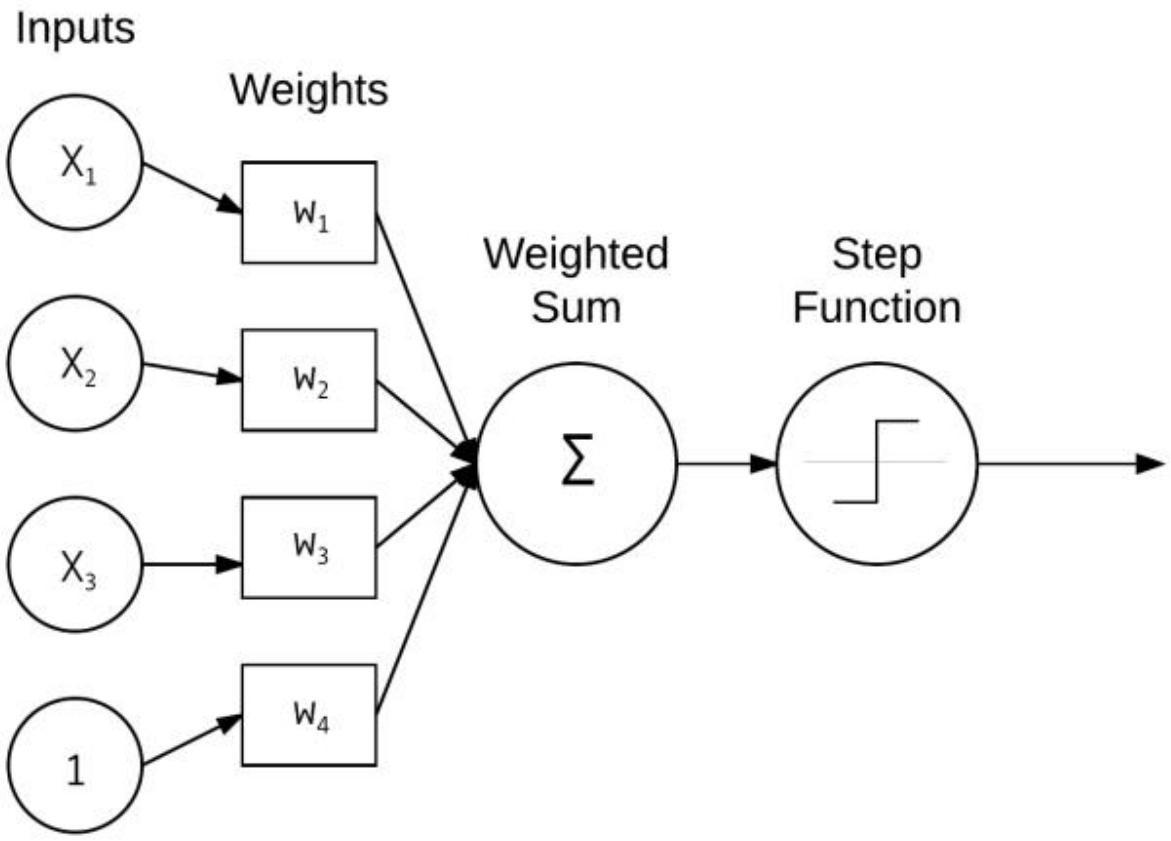

Figure 1: A simple NN that takes the weighted sum of the input x and weights w.

In practice these inputs could be vectors used to quantify the contents of an image in a systematic, predefined way (e.x., color histograms, Histogram of Oriented Gradients, Local Binary Patterns, etc.). In the context of deep learning, these inputs are the *raw pixel intensities* of the images themselves.

Each *x* is connected to a neuron via a weight vector ***W*** consists of *w*1*; w*2*; ::: wn*, meaning that for each input *x* we also have an associated weight *w*. Finally, the *output node* on the right of Figure 1 takes the weighted sum, applies an activation function *f* (used to determine if the neuron "fires" or not), and outputs a value. Expressing the output mathematically, you'll typically encounter the following three forms:

• $f\,(w1x1 + w2x2 + \cdots + wnxn)$

• $f\,(\sum n\ i{=}1\ wixi)$

• Or simply, f (net),

where $net = \sum_{i=1}^{n} w_i x_i$ Regardless how the output value is expressed, understand that we are simply taking the weighted sum of inputs, followed by applying an activation function f .

1.1 Multi-layer Networks with Keras

Keras is a dedicated (highly optimized) neural network library and therefore the preferred implementation method for multi-layer neural networks.

1.2 Important Parameters in a Neural Network [1]

There are four main ingredients you need to put together in your own neural network and deep learning algorithm:

- **Dataset**

The dataset is the first ingredient in training a neural network – the data itself along with the problem we are trying to solve define our end goals.

Usually, we have little choice in our dataset– we are given a dataset with some expectation on what the results from our project should be. It is then up to us to train a machine learning model on the dataset to perform well on the given task.

- **Loss Function**

Given our dataset and target goal, we need to define a loss function that aligns with the problem we are trying to solve. In nearly all image classification problems using deep learning, we'll be using cross-entropy loss. For > 2 classes we call this *categorical cross-entropy*. For two class problems, we call the loss *binary cross-entropy*.

- **Model/Architecture**

Your network architecture can be considered the first actual "choice" you have to make as an ingredient. Your dataset is likely chosen for you (or at least you've *decided* that you want to work with a given dataset). And if you're performing classification, you'll in all likelihood be using cross-entropy as your loss function. However, your network architecture can vary dramatically, especially when with which optimization method you choose to train your network. When deciding about network architecture, consider following:

1. How many data points you have.

2. The number of classes.

3. How similar/dissimilar the classes are.

4. The intra-class variance.

The number of layers and nodes in your network architecture (along with any type of regularization) is likely to change as you perform more and more experiments. The more results you gather, the better equipped you are to make informed decisions on which techniques to try next.

- ### Optimization Method

The final ingredient is to define an optimization method. As we've seen thus far in this book *Stochastic Gradient Descent* is used quite often.

Even despite all these newer optimization methods, SGD is *still* the workhorse of deep learning – most neural networks are trained via SGD, including the networks obtaining state-of-the-art accuracy on challenging image datasets such as ImageNet.

When training deep learning networks, especially when you're first getting started and learning the ropes, *SGD should be your optimizer of choice*.

2. Strategies to Prevent Overfitting [2]

Overfitting is a common challenge in training neural networks. It is the behavior of *decreasing* training loss while validation loss *increases.*

Here are the most common ways to prevent overfitting in neural networks:

1. Getting more training data.
2. Descending Learning Rate
3. Reducing the capacity of the network.
4. Adding weight regularization.
5. Adding dropout.
6. Data-augmentation
7. Batch normalization

First option results in heavier computation which is not possible for my operating system. Therefore, I have implemented 2, 3, 4, and 5.

2.1 Descending Learning Rate

Many models train better if you gradually reduce the learning rate during training. The code below sets a schedules.InverseTimeDecay to hyperbolically decrease the learning rate to 1/2 of the base rate at 1000 epochs, 1/3 at 2000 epochs and so on (Figure 2).

```
lr_schedule = tf.keras.optimizers.schedules.InverseTimeDecay(
  0.001,
  decay_steps=STEPS_PER_EPOCH*1000,
  decay_rate=1,
  staircase=False)

optimizers= SGD(lr_schedule)
```

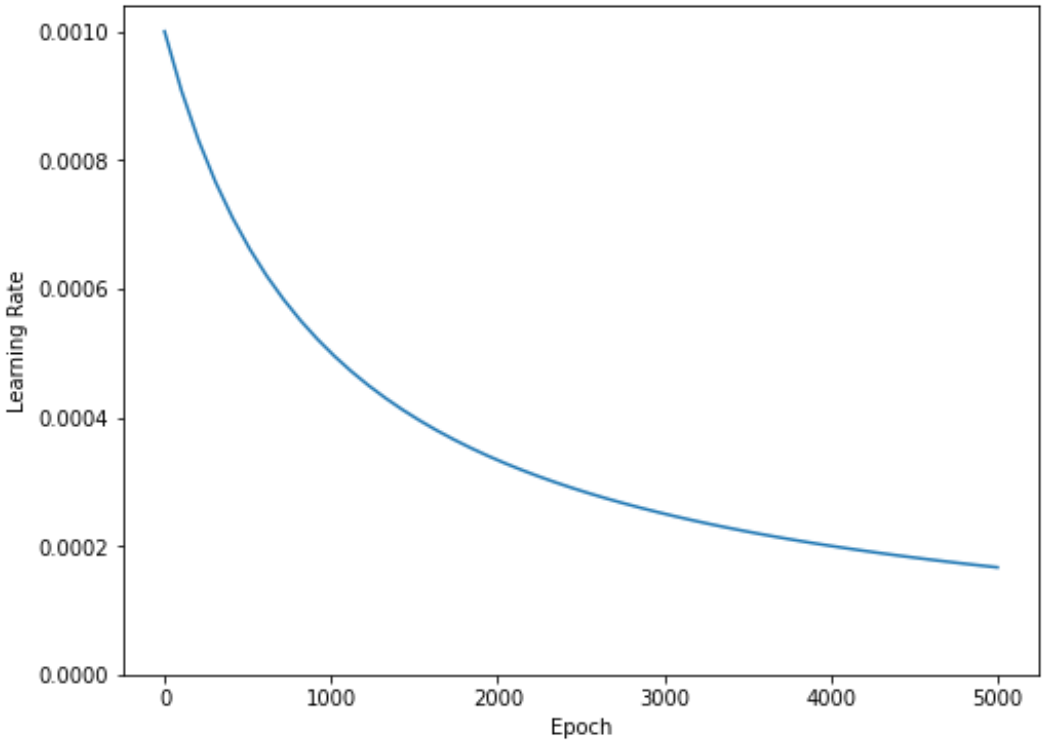

Figure 2: Descending Learning Rate

2.2 Adding weight regularization

You may be familiar with Occam's Razor principle: given two explanations for something, the explanation most likely to be correct is the "simplest" one, the one that makes the least amount of assumptions. This also applies to the models learned by neural networks: given some training data and a network architecture, there are multiple sets of weights values (multiple models) that could explain the data, and simpler models are less likely to overfit than complex ones.

A "simple model" in this context is a model where the distribution of parameter values has less entropy (or a model with fewer parameters altogether, as we saw in the section above). Thus a common way to mitigate overfitting is to put constraints on the complexity of a network by forcing its weights only to take small values, which makes the distribution of weight values more "regular". This is called "weight regularization", and it is

done by adding to the loss function of the network a cost associated with having large weights. This cost comes in two flavors:

- L1 regularization, where the cost added is proportional to the absolute value of the weights coefficients (i.e. to what is called the "L1 norm" of the weights).
- L2 regularization, where the cost added is proportional to the square of the value of the weights coefficients (i.e. to what is called the squared "L2 norm" of the weights). L2 regularization is also called weight decay in the context of neural networks. Don't let the different name confuse you: weight decay is mathematically the exact same as L2 regularization.

L1 regularization pushes weights towards exactly zero encouraging a sparse model. L2 regularization will penalize the weights parameters without making them sparse since the penalty goes to zero for small weights: one reason why L2 is more common.

In `tf.keras`, weight regularization is added by passing weight regularizer instances to layers as keyword arguments. Let's add L2 weight regularization now.

```
l2_model = tf.keras.Sequential([
  layers.Dense(512, activation='elu',
               kernel_regularizer=regularizers.l2(0.001),
               input_shape=(FEATURES,)),
  layers.Dense(512, activation='elu',
               kernel_regularizer=regularizers.l2(0.001)),
  layers.Dense(512, activation='elu',
               kernel_regularizer=regularizers.l2(0.001)),
  layers.Dense(512, activation='elu',
               kernel_regularizer=regularizers.l2(0.001)),
  layers.Dense(1)
])
```

`l2(0.001)` means that every coefficient in the weight matrix of the layer will add `0.001 * weight_coefficient_value**2` to the total **loss** of the network.

2.3 Adding dropout

Dropout is one of the most effective and most commonly used regularization techniques for neural networks, developed by Hinton and his students at the University of Toronto.

The intuitive explanation for dropout is that because individual nodes in the network cannot rely on the output of the others, each node must output features that are useful on their own.

Dropout, applied to a layer, consists of randomly "dropping out" (i.e. set to zero) a number of output features of the layer during training. Let's say a given layer would normally have returned a vector [0.2, 0.5, 1.3, 0.8, 1.1] for a given input sample during training; after applying dropout, this vector will have a few zero entries distributed at random, e.g. [0, 0.5, 1.3, 0, 1.1].

The "dropout rate" is the fraction of the features that are being zeroed-out; it is usually set between 0.2 and 0.5. At test time, no units are dropped out, and instead the layer's output values are scaled down by a factor equal to the dropout rate, so as to balance for the fact that more units are active than at training time.

In tf.keras you can introduce dropout in a network via the Dropout layer, which gets applied to the output of layer right before.

Let's add two Dropout layers in our network to see how well they do at reducing overfitting:

```
dropout_model = tf.keras.Sequential([
    layers.Dense(512, activation='elu', input_shape=(FEATURES,)),
    layers.Dropout(0.5),
    layers.Dense(512, activation='elu'),
    layers.Dropout(0.5),
    layers.Dense(512, activation='elu'),
    layers.Dropout(0.5),
    layers.Dense(512, activation='elu'),
    layers.Dropout(0.5),
    layers.Dense(1)
])

regularizer_histories['dropout'] = compile_and_fit(dropout_model, "regularizers/dropout")
```

It's clear from this plot that both of these regularization approaches improve the behavior of the "Large" model. But this still doesn't beat even the "Tiny" baseline.

2.4 Combined L2 + dropout

Next try L2 regularizer and dropout both, together, and see if that does better.

```
combined_model = tf.keras.Sequential([
    layers.Dense(512, kernel_regularizer=regularizers.l2(0.0001),
                 activation='elu', input_shape=(FEATURES,)),
    layers.Dropout(0.5),
    layers.Dense(512, kernel_regularizer=regularizers.l2(0.0001),
                 activation='elu'),
    layers.Dropout(0.5),
    layers.Dense(512, kernel_regularizer=regularizers.l2(0.0001),
                 activation='elu'),
    layers.Dropout(0.5),
    layers.Dense(512, kernel_regularizer=regularizers.l2(0.0001),
                 activation='elu'),
    layers.Dropout(0.5),
    layers.Dense(1)
])
```

3. Implementation, Results and Discussion

3.1 Modified Parameters

The implementation is done in following steps using the base code in [1]:

1) Load Data

Cifar-10 dataset was used in this assignment. The dataset has 60,000 images which was too heavy for my CPU. Therefore, I have used 12,000 images from this dataset.

2) Define Keras Model

Following models have been tried:

Model Architecture		Weight regularization & Dropout
4-layer model 3072 x 1024 x 512 x 10		None
	X	weight regularization (L2)
3-layer model 3072 x 1024 x 10		weight regularization (L2) + dropout

3) Compile Keras Model

Compiling was done with following options:

Loss Function		Optimizer		Learning Rate		Momentum
categorical_ crossentropy	X	SGD	X	Constant (0.01)	X	0
				Descending		0.5

4) Fit Keras Model

Model is fit using training and validated using testing data.

5) Evaluate Keras Model

Evaluation is done using following options:

- Testing data from Cifar-10 dataset (2,000 images)
- Unknown truck images from Internet passed through a filter (scale, blur, flip, rotate) (167 images)

Unknown truck images have been passed through 6 different filters resulting in 8 following test batches (including Cifar-10 test batch for comparison):

1) Truck images Cifar-10 test batch
2) Unknown truck images (original)
3) Unknown truck images (scaled)
4) Unknown truck images (blurred)
5) Unknown truck images (flipped horizontally)

6) Unknown truck images (flipped vertically)
7) Unknown truck images (rotated 180 degree)
8) Unknown truck images (rotated 90 degree)

3.2 Results

3.2.1 Training Loos and Accuracy for Base Model

Figure 3: Training Loos and Accuracy for Base Model

As illustrated in figure 3:

- Loss
  - Training loss is constantly decreasing but validation loss is first decreasing and then starts increasing; (Overfitting)
- Accuracy
  - While training accuracy is reaching nearly 100%, validation accuracy is less than 50%.

3.2.2 The Effect of Varying Model Parameters

The following table summarizes the defining parameters for models and optimizers and the resulting accuracies:

Table 1: Models' Parameters (Colors match Figures 4, 5, 6, 7)

Run#	Dataset Size (Train+Test)	Loss Function	Model/ Architecture	Optimization (Learning Rate)	Weight Regularization	Accuracy (Average)*
[0]	50k+10k	categorical_ crossentropy	4-Layer	SGD(0.01)*	None	
[1]	10k + 2k	categorical_ crossentropy	3-Layer	SGD(0.01)	None	45%
[2]	10k + 2k	categorical_ crossentropy	3-Layer	SGD(0.01)	L2	44%
[3]	10k + 2k	categorical_ crossentropy	3-Layer	SGD(0.01)	L2+dropout	47%
[4]	10k + 2k	categorical_ crossentropy	3-Layer	SGD(0.01*)	None	46%
[5]	10k + 2k	categorical_ crossentropy	3-Layer	SGD(0.01*)	L2	46%
[6]	10k + 2k	categorical_ crossentropy	3-Layer	SGD(0.01*)	L2+dropout	48%
[7]	10k + 2k	categorical_ crossentropy	3-Layer	SGD(0.01, Momentum(0.5))	None	46%
[8]	10k + 2k	categorical_ crossentropy	3-Layer	SGD(0.01, Momentum(0.5))	L2	45%
[9]	10k + 2k	categorical_ crossentropy	3-Layer	SGD(0.01, Momentum(0.5))	L2+dropout	46%
[10]	10k + 2k	categorical_ crossentropy	3-Layer	SGD(0.01*, Momentum(0.5))	None	46%
[11]	10k + 2k	categorical_ crossentropy	3-Layer	SGD(0.01*, Momentum(0.5))	L2	46%
[12]	10k + 2k	categorical_ crossentropy	3-Layer	SGD(0.01*, Momentum(0.5))	L2+dropout	48%
[13]	10k + 2k	categorical_ crossentropy	4-Layer	SGD(0.01)	None	46%
[14]	10k + 2k	categorical_ crossentropy	3-Layer	SGD(0.01*, Momentum(0.5))	None	48%

[15]	10k + 2k	categorical_ crossentropy	3-Layer	SGD(0.01*, Momentum(0.5))	L2	46%
[16]	10k + 2k	categorical_ crossentropy	3-Layer	SGD(0.01*, Momentum(0.5))	L2+dropout	48%

Note:

- SGD (α): SGD with learning rate= α
- SGD (α*): SGD with descending learning rate, starting from α
- Momentum(γ): Momentum term = γ
- Accuracy (Average)*= Average accuracies for epochs=81, 82, …, 100

3.2.3 Validation Accuracy for 3-Layer Models

In order to examine the effect of different parameters on the accuracy of NN, I have trained 12 3-layer models using the parameters in table.

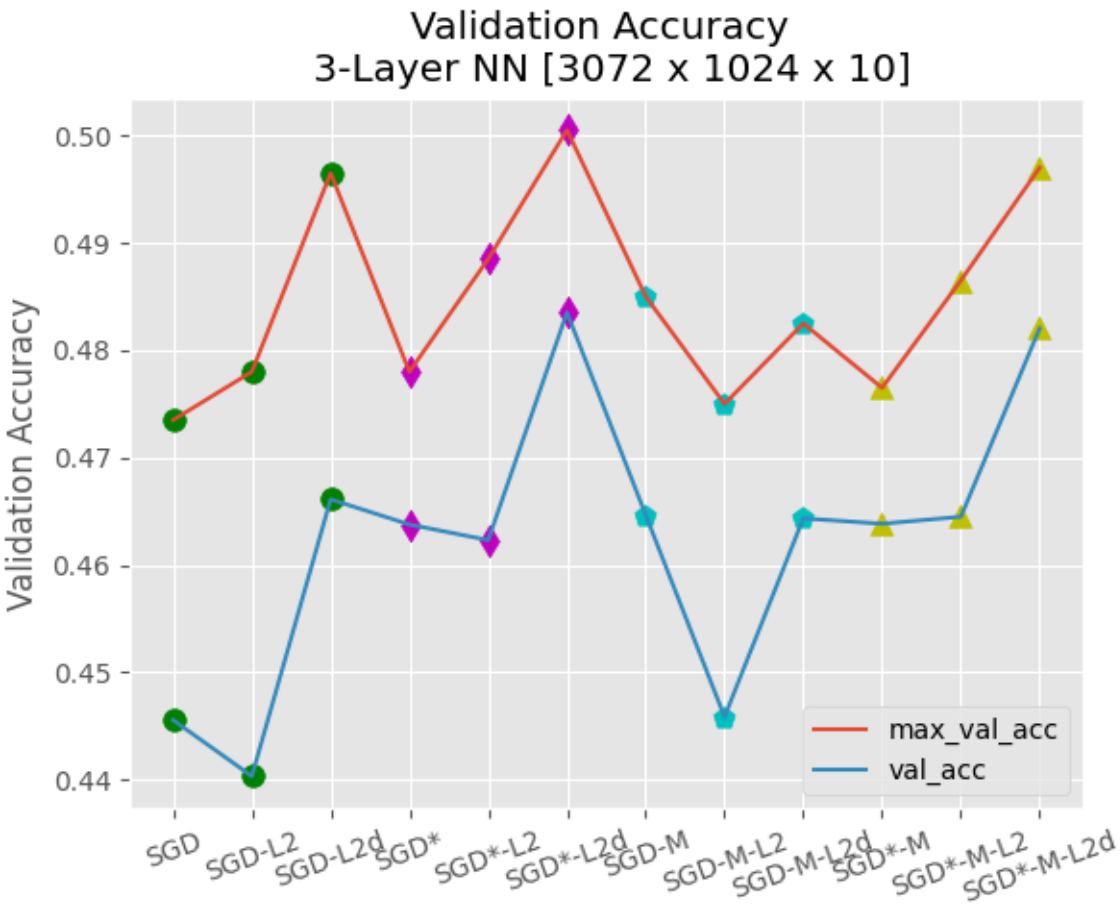

Figure 4: Validation Accuracy for 3-Layer Models

As illustrated in figure 4:

- Validation accuracies for models with same optimizer (but different weight regularizer) are displayed with the same color and marking
- Effect of weight regularization:
  - L2 weight regularization reduces average validation accuracy but increases max validation accuracy
  - However L2d (L2 + dropout) significant increases both average and max validation accuracies
- Effect of learning rate and momentum:
  - There is not a clear pattern for the effect of momentum;
  - However, SGD* and SGD*-M (descending learning rate with/without momentum) are slightly better than SGD and SGD-M (constant learning rate with/without momentum).

3.2.4 Training & Validation Loss for 3-Layer Models

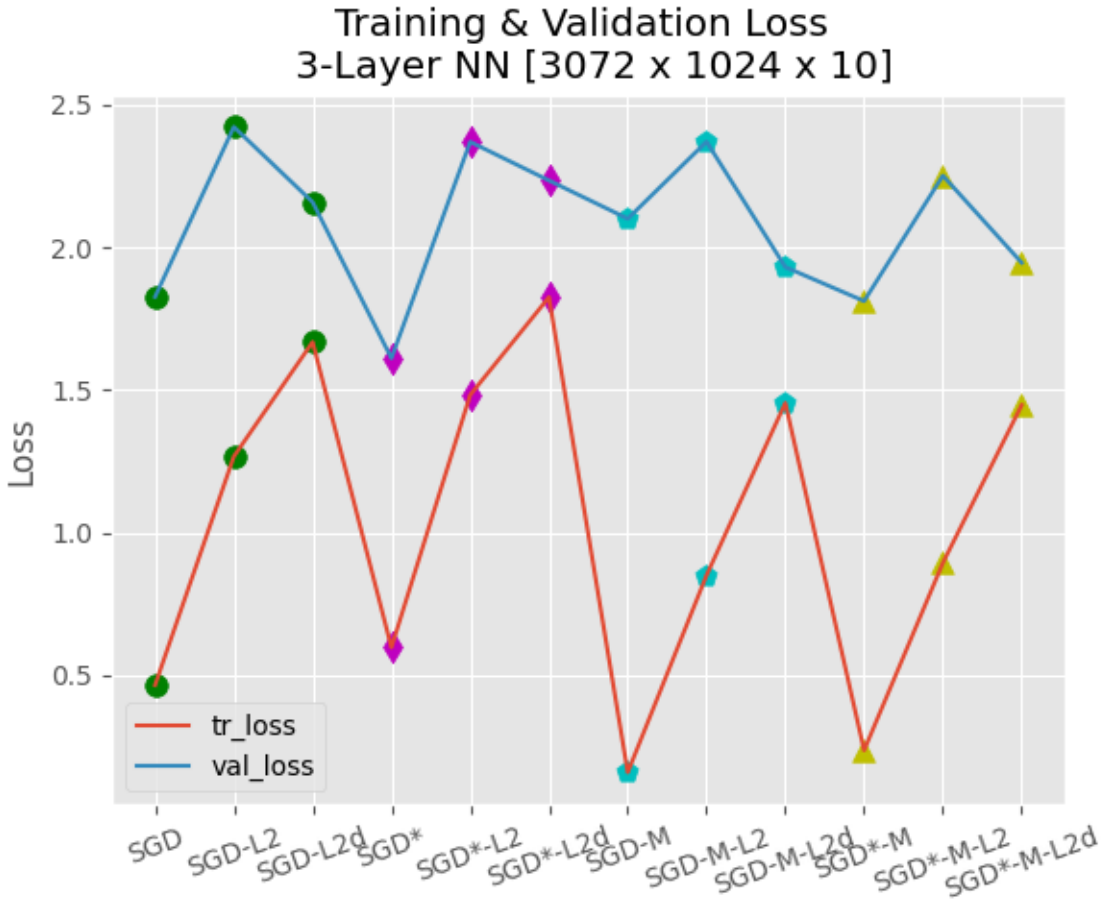

Figure 5: Training & Validation Loss for 3-Layer Models

As illustrated in figure 5:

- Training & validation loss for models with same optimizer (but different weight regularizer) are displayed with the same color and marking
- Effect of weight regularization:
  - L2 weight regularization increases both training & validation loss

- However, L2d (L2 + dropout) increases training loss but reduces validation loss compared to L2

- Effect of learning rate and momentum:
  - There is not a clear pattern for the effect of momentum;

3.2.5 Validation Accuracy for 3-Layer Models vs. 4-Layer Models

In order to examine the effect of architecture on the accuracy of NN, I also have trained 4-layer models using the parameters in table.

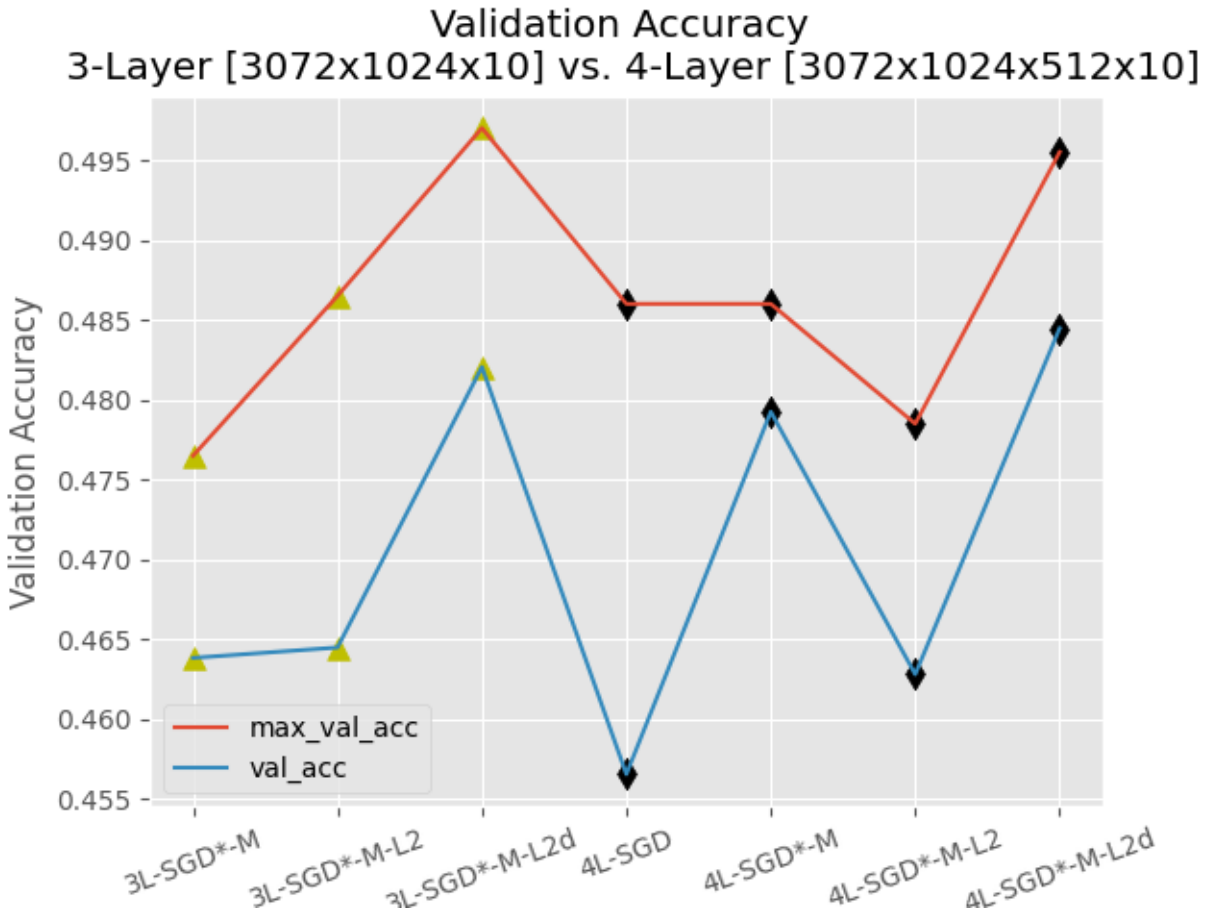

Figure 6: Validation Accuracy for 3-Layer Models vs. 4-Layer Models

As illustrated in figure 6:

- Validation accuracies for models with same architecture (but different weight regularizer) are displayed with the same color and marking
- Effect of parameter modification
  - 4L-SGD points are the average and max accuracies for base model (default parameters)
  - The average accuracy of all modified models (3-layer & 4-layer) is higher than base model;
  - There is no specific pattern for max accuracies
- Effect of weight regularization for 4-Layer models:
  - L2d (L2 + dropout) significant increases both average and max validation accuracies

- Effect of model architecture
    - 4-layer models have accuracies slightly higher than 3-layer models but the difference is less than expected

3.2.6 Training & Validation Loss for 3-Layer Models vs. 4-Layer Models

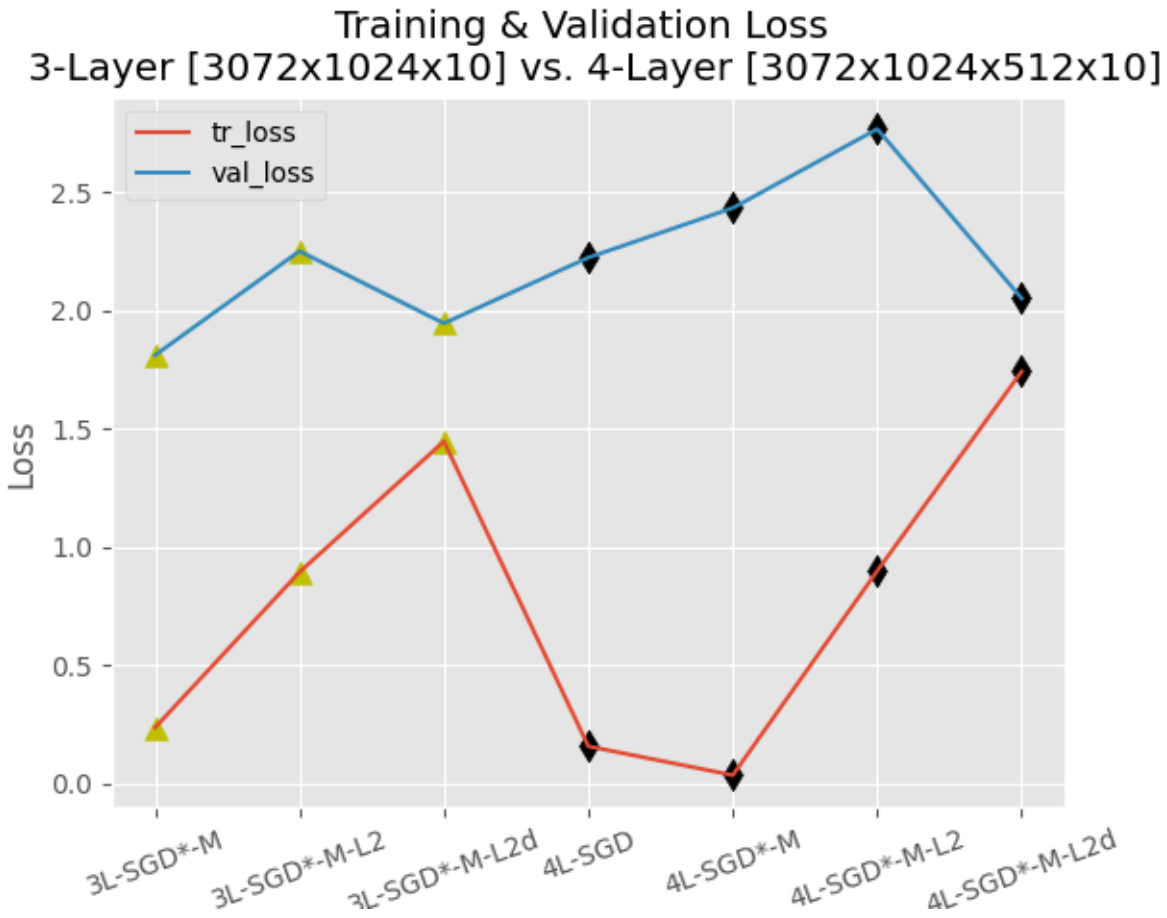

Figure 7: Training & Validation Loss for 3-Layer Models vs. 4-Layer Models

As illustrated in figure 7:

- Training & validation loss for models with same optimizer (but different weight regularizer) are displayed with the same color and marking
- Effect of weight regularization:
    - L2 weight regularization increases both training & validation loss
    - The combination L2d (L2 + dropout) increases training loss but reduces validation loss compared to L2
- Effect of learning rate and momentum:
    - There is not a clear pattern for the effect of momentum;

3.2.7 Classification Precision for Trucks for Two Models

This section represents the accuracy of trained model when used for classification of images outside dataset.

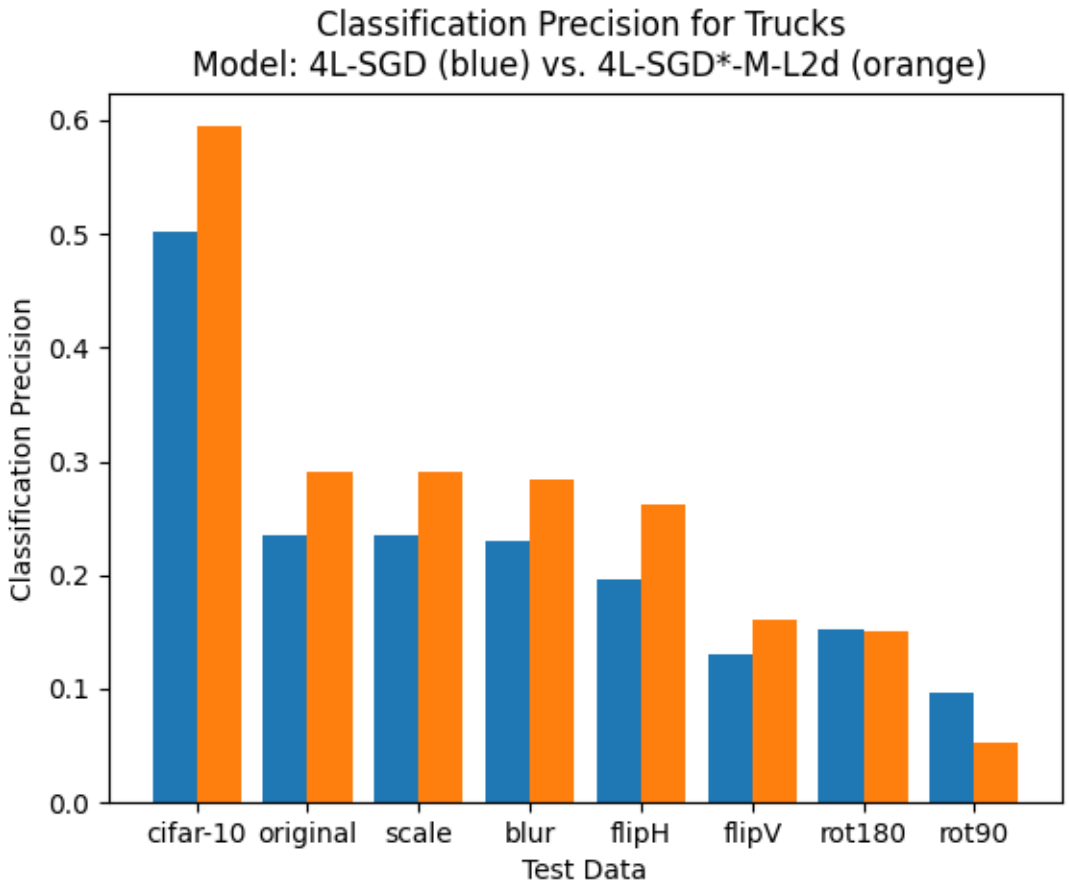

Figure 8: Classification Precision for Trucks for Two Models

As illustrated in figure 8:

- Dataset vs. unknown image:
  - Although test dataset is only used for validation, and not for training, their classification accuracy is about 50%: almost 2 time more than unknown images
- Base model (4L-SGD) vs. modified model (4L-SGD*-M-L2d)
  - Almost for all test batches, modified model has higher accuracies than base model
- Effect of filters
  - Scale has no effect as all images are scaled to 32x32x3 in pre-processing step (30%)
  - Blur and horizontal flip have very little effects (30%)
  - Vertical flip and 180-degree rotation are very similar with less than 20% accuracies
  - Among all filters, 90-degree rotation has most effect, reducing accuracy from 30% to less than 10%

3.2.8 Discussion

Overfitting is a common problem when training neural networks. Four different strategies were tried for avoiding this problem and increasing validation accuracy.

- Effect of weight regularization:
  - L2 weight regularization reduces average validation accuracy but increases max validation accuracy
  - L2d (L2 + dropout) significant increases both average and max validation accuracies
  - L2 weight regularization increases both training & validation loss
  - L2d (L2 + dropout) increases training loss but reduces validation loss compared to L2

- Effect of learning rate and momentum:
  - There is not a clear pattern for the effect of momentum;
  - However, SGD* and SGD*-M (descending learning rate with/without momentum) result in validation accuracies slightly higher than SGD and SGD-M (constant learning rate with/without momentum).
- Effect of model architecture
  - 4-layer models have accuracies slightly higher than 3-layer models but the difference is less than expected
- Overall Effect of parameter modification
  - 4L-SGD points are the average and max accuracies for base model (default parameters)
  - The average accuracy of all modified models (3-layer & 4-layer) is higher than base model;
  - There is no specific pattern for max accuracies

Finally it is worth mentioning that more tests are still required for ensuring the repeatability of these conclusions which would take long time (each run lasts about 40 minutes).

References

[1] book 'Deep Learning for Computer Vision - Starter Bundle' by Adrian Rosebrock
[2] https://www.tensorflow.org/tutorials/keras/overfit_and_underfit