Machine Learning Practical: Coursework 4

s1667278

16 March 2017

1 Introduction

This paper will experiment with various aspects of a convolutional neural network (CNN)[1] to predict labels in the CIFAR-10 and CIFAR-100 data. The goal of the paper will be to determine how, why, and to what degree tweaking aspects of the model affects performance, speed of convergence, run time, and other factors. The goal of the paper will *not* be to design a model that gives the best test error and accuracy, since this would require building a deep, complex network that would not provide a good sandbox for experimentation.

This paper will use the final model from Coursework 3 as the baseline with which the new architecture will be compared [1]. The process will be iterative: Each time a new aspect of the model is tested, the new information will be incorporated into the model and used as the basis for the next experiment. An aspect of the model may be carried over to the next experiment if it is likely improve the quality of experiments going forward.[2]

As in Coursework 3, "[m]ost experiments will use softmax cross entropy error instead of accuracy as a measure of performance, since it is arguably more nuanced and comprehensive. Accuracy will be used mostly as a means of getting an intuitive sense of how well the model is doing" [1].

The aspects of the model that will be tweaked are: batch normalization; dropout rate; kernel size and stride size of the convolutional layers; and window size and stride size of the pooling layers.

1.1 Description of data

Classification of the CIFAR data is a common task in machine learning—and in image recognition especially [2].

The CIFAR-10 data consists of 60,000 32×32 images. There are 10 mutually exclusive class labels, with 6,000 images of each label. The class labels are *airplane*, *automobile*, *bird*, *cat*, *deer*, *dog*, *frog*,

[1] A convolutional neural network is a type of neural network that exploits strong local correlations between input variables and is typically used on two-dimensional image data.

[2] This *may* include performance improvements, but not at the expense of greatly increased complexity or run time.

horse, *ship*, and *truck*. The data is split into 50,000 images for training and 10,000 for testing. See 8 in the Appendix for examples images in each of the 10 classes [3].

The CIFAR-100 data consists of 20 "superclasses" containing 5 class labels, for a total of 100 class labels.[3] For example, one superclass is *fish*, and it contains the class labels *aquarium fish*, *flatfish*, *ray*, *shark*, and *trout*. See the Appendix for a description of these classes and their superclasses [3].

At the end of the analysis, the final model and hyperparameter settings will be used on the CIFAR-100 for the sake of comparison. The experiments will not be conducted on both CIFAR-10 and CIFAR-100 throughout the paper in an effort to avoid excess complexity during experimentation.

1.2 Overview baseline architecture from Coursework 3

The goal of Coursework 3 was to provide a model with reasonable performance that could be used as a baseline for this coursework. This architecture has one input layer, four hidden layers with 100 hidden units each, and one output layer, with ReLU nonlinearities wrapping the affine transformations. The model uses a truncated normal weight initialization with a standard deviation of $2/\sqrt{input\_dim + output\_dim}$. Biases are initialized with zeros. For the learning schedule, the model used Adagrad with a learning rate of 0.01. The model was trained using multi-class cross entropy error with softmax with batches of 50 images [1].

2 Design of the CNN

See the Appendix for research conducted on typical CNN architecture designs in both instructional and state-of-the-art implementations. Instructional examples are those use for teaching purposes—e.g., examples on the TensorFlow website—while state-of-the-art implementations are those that have performed very well in image recognition competitions.

A common architecture pattern for many CNNs is

```
INPUT -> [[CONV -> RELU]*N -> POOL?]*M -> [FC -> RELU]*K -> FC
```

where the `*` represents repetition, `INPUT` is the input layer, `CONV` is a convolutional layer, `RELU` is a ReLU layer, `POOL?` is an optional pooling layer, and `FC` is a fully connected layer. Typically, $0 \geq$ `N` ≥ 3 and $0 \geq$ K > 3. In order for the model to qualify as a CNN, `M` > 0. [4].

For the purpose of this paper, a model was chosen to have just enough complexity to test different aspects of a CNN, but not so much complexity that it was difficult to generalize the results. For this reason, the experiments will use a model closer to those used in instructional examples as opposed to state-of-the-art models.

[3] All analysis will be conducted using the 100 class labels and not the superclasses.

In instructional examples, using two convolutional layers ($\texttt{M} = 2$) was the most popular choice. This seems an appropriate choice for this paper since it would allow experiments to explore whether some model changes had more of an effect on the convolutional layers closer or further from the inputs. At the end of both instructional and state-of-the-art networks, there are often a few fully connected layers. Again, this seems reasonable.

This paper will therefore use an architecture with $\texttt{N} = 1$, $\texttt{M} = 2$ and $\texttt{K} = 2$:

```
INPUT -> [[CONV -> RELU]*1 -> POOL]*2 -> [FC -> RELU]*2 -> FC
```

The model will start with a kernel size of 3×3, since this is something easy that we can replicate from the state-of-the-art models like VGGNet [5]. (The instructional models tend to use 5×5.) Pooling layers with 2×2 seems most popular, both in instructional and state-of-the-art. The model will use the number of units in the fully connected layers from TensorFlow's CIFAR-10 example [2].

Expanding on the notation from earlier, this gives the network

$$
\begin{aligned}
&\texttt{INPUT}_{32\times32\times1} \texttt{ ->} \\
&\texttt{CONV(3, 1)}_{filters=32} \texttt{ -> RELU -> POOL(2, 2) ->} \\
&\texttt{CONV(3, 1)}_{filters=64} \texttt{ -> RELU -> POOL(2, 2) ->} \\
&\texttt{FC}_{384} \texttt{ -> RELU ->} \\
&\texttt{FC}_{192} \texttt{ -> RELU ->} \\
&\texttt{FC}_{10}
\end{aligned}
$$

where subscripts indicate the output dimensionality of an image, $\texttt{CONV(}w\texttt{,}s\texttt{)}$ indicates a convolutional layer with width (and height) w and stride size s, and $\texttt{POOL(}w\texttt{,}s\texttt{)}$ indicates a pooling layer with width (and height) w and stride size s.[4] Padding is always used in both the convolutional and pooling layers [6, 7]. After the final pooling layer, there is a layer that flattens the tensor. This flattened layer feeds into a dropout layer, and the dropout layer then feeds into the first fully connected layer. The model uses an Adam learning schedule with a learning rate of 10^{-4}. The weights are initialized using TensorFlow's `tf.truncated_normal` function, and the biases using `tf.constant` with a value of 0.1.

This CNN model performs better than the Coursework 3 baseline, achieving a validation error of 1.14 and an accuracy of 0.62—which is higher than the baseline's 1.42 and 0.51. The run time is 2.25 minutes per epoch, which is much more than the baseline of around 7 seconds per epoch.

[4] There is some implied dimensionality with this notation. For example, activation layers like ReLU keep the dimensionality of the previous layer, so there are no subscripts. With convolutional layers, the dimensionality is implied by the number of output filters. In the case of the first convolutional layer, the implied dimensionality of the output is $28 \times 28 \times 32$. The implied dimensionality of the first pooling layer is $14 \times 14 \times 32$ since $s = 2$.

3 Batch normalization

Batch normalization is a technique used to fix "internal covariate shift", which is the shifting of the distribution of layer inputs during training. This internal covariate shift makes training large networks difficult mainly because it slows down training.

Batch normalization is known to be beneficial in more straightforward feedforward networks, with reported benefits including regularization and faster learning [8]. This was seen in Coursework 2, where "[c]onvergence [was] much faster for most of the batch normalization models" when compared to models where batch normalization was not implemented [9].[5]

The authors of the original batch normalization research claim that "by only using Batch Normalization (BN-Baseline), [they were able to] match the accuracy of Inception in less than half the number of training steps" [8]. However, this may not be the whole truth, since computational time—"wall time"—is not considered when counting the number of training steps. After all, machine learning practitioners care about the time it takes for a model to converge rather than the number of training steps. Therefore, it is worth paying special attention to the actual amount of time to convergence when evaluating batch normalization with the convolutional layers [10].

This experiment will therefore test the original claim of the authors of the batch normalization paper to see whether CNN models converge faster when batch normalization is applied to the convolutional layers—but the experiment will take wall time into account when assessing speed of convergence. This is the first experiment because it is expected to improve time to convergence, and the results can be used on later experiments.

3.1 Description of experiment

There are two boolean switches in the model: one for batch normalization in *both* fully connected layers, and one for batch normalization in *both* convolutional layers. This experiment will test all four combinations of fully-connected layer batch normalization in $\{True, False\}$ and convolutional layer batch normalization in $\{True, False\}$. The implementation below provides more details.

3.2 Implementation

The variables described above and their boolean values are represented as key-value pairs in a dictionary of hyperparameters `params`. This dictionary is passed into the main function that runs the model. For the moment, dictionary entries not relevant to this experiment will be replaced with ellipses.

```
...
params = {
    ...  # other hyperparameters
```

[5] However, batch normalization, in its implementation in Coursework 3, did not improve convergence or regularization. However, this could be due to either the particular faulty implementation.

```
    'fc_batch_norm': None,  # to be assigned in loop
    'conv_batch_norm': None,  # to be assigned in loop
    ...  # other hyperparameters
}
...
combos = [(True, True), (False, True), (True, False), (False, False)]  # (fc, conv)
for combo in combos:
    ...
    params['fc_batch_norm'] = combo[0]
    params['conv_batch_norm'] = combo[1]
    run_cnn_model(params)
    ...
...
```

Within `run_cnn_model`, the boolean values `params['fc_batch_norm']` and `params['conv_batch_norm']` are then passed as arguments into the wrapper functions `fc_layer` and `conv_layer`, respectively. This boolean triggers the wrapper function `batch_normalization` in each of those functions; this function applies batch normalization to the processed layer before it is output. For example, the boolean passed in as the `batch_norm` argument in `fc_layer` as:

```
def fc_layer(x, ..., batch_norm):
    ...
    layer = ...  # define layer using input data x and other arguments
    if batch_norm:
        layer = batch_normalization(layer)
    ...
    return layer
```

For flattened layers, the TensorFlow function `tf.nn.batch_normalization` is used in the `batch_normalization` function. For 4-D tensors, `tf.nn.lrn` is used. See the Appendix for the code for `batch_normalization`.

3.3 Results and comments

The effects of batch normalization (figure 1) can be decomposed into a few separate effects:

- **Convergence speed.** When applied to fully connected layers, batch normalization results in much faster convergence at nearly the same run time ("wall time") per epoch (figure 2). However, convergence speed is not increased when batch normalization is applied to the convolutional layers. In fact, convergence is *slower* in terms of number of epochs *as well as run time*, which increased by 62 or 83% (depending on whether the fully connected layers also have been batch normalized) when the convolutional layers are batch normalized (figure 2).
- **Validation error.** The minimum validation error across epochs is lowest when batch normalization is applied to the fully connected layers (with values around 1.16).[6] However, as mentioned earlier, the preference between those two models still goes to the one where the convolutional layer is not batch normalized, due to its slower run time.

[6]The last epoch of the run without any batch normalization has a validation error of 1.14, but this seems to be noise and is not consistent across epochs.

- **Regularization.** The regularization effects are not apparent when applied to the fully connected layers. On the models where the fully connected layers were batch normalized, there is serious overfitting past epoch 10.[7] When applied to the convolutional layers, it actually seems make overfitting worse.

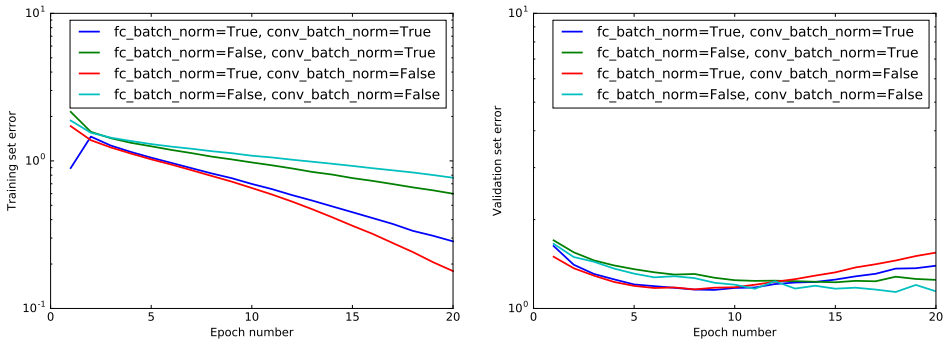

Figure 1: Training and validation error for batch normalization.

Batch normalization on...		*Run time*
FC layers	*Conv layers*	*(min)*
True	True	82.1
False	True	75.9
True	False	44.9
False	False	46.9

Figure 2: Run time of batch normalization experiment.

In short, batch normalization seems to be helpful when applied to fully connected layers in our sandbox model, but its major effect when applying to convolutional layers is to increase run time per epoch—at least in the implementation used in this paper.[8] There is also overfitting when fully connected layers were batch normalized. Although batch normalization has proven helpful in a number of state-of-the-art applications in the past few years, the claimed effects in the seminal paper—regularization and improved convergence—may not be applicable to all situations.

[7] However, it is unclear whether overfitting would have been worse if there was *no* batch normalization but the learning rate was increased so that convergence occurred somewhere between epoch 5 and 10.

[8] In a more naive implementations of batch normalization—where 4-D tensors are flattened, normalized, and then reshaped to the original size—run times were even higher: around 10 minutes per epoch instead of around 2 when convolutional layers were not batch normalized and around 4 when they were batch normalized using `tf.nn.lrn`.

3.4 Model changes

Going forward, batch normalization will be applied to the fully connected layers but not to the convolutional layers for reasons stated in this section.

4 Dropout

Dropout is randomly zeroing out output values in order to simulate the behavior of an average of multiple neural network models. The primary hyperparameter to tweak is the probability p that any unit is "dropped", usually with $p \sim 0.5$ [11]. Its primary benefit is regularization, although training time can take 2–3 times longer than a network of the same architecture [11, 12].

This experiment's goal is to explore the effects of a single dropout layer against the recommended implementation of multiple layers across the architecture—as well as to curb overfitting in future experiments. It will be broken down into two parts:

- First, **different learning rates will be tested on dropout settings similar to those used on the CIFAR-10 data in the original dropout paper**. In these settings, there are there are multiple dropout layers spread throughout the architecture. This makes sense, since the robustness introduced through regluarization is applied at multiple levels of feature abstraction and therefore is more likely to make the model more robust overall. In other words, early dropout layers may make the network robust to noise in the inputs, but not robust to entire high-level features missing. On the other hand, dropout layers at the end of the network may be less robust to small initial perturbations, but more robust to missing or varying high-level features.
- Second, **a single dropout layer with varying** p will be tested to see if convergence speed is improved relative to the more complete implementation dropout from the first part of the experiment. The single layer will be placed in the middle of the network in order to compromise robustness at different levels of abstraction.

4.1 Description of experiment

4.1.1 Iterating over learning rate

In the initial paper on dropout, "we use p = (0.9, 0.75, 0.75, 0.5, 0.5, 0.5) going from the input layer to the top" [12]. In order to simplify implementation a bit, this part of the experiment will place one layer immediate after the inputs with $p = 0.8$ and before every fully connected layer with $p = 0.5$. The architecture is therefore

$\texttt{INPUT}_{32\times32\times1}$ ->
DROP(0.8) ->

```
CONV(3, 1)_{filters=32} -> RELU -> POOL(2, 2) ->
CONV(3, 1)_{filters=64} -> RELU -> POOL(2, 2) ->
DROP(0.5) -> FC_384 -> RELU ->
DROP(0.5) -> FC_192 -> RELU ->
DROP(0.5) -> FC_10
```

where `DROP(`p`)` is the dropout layer with dropout probability p.

Given this fixed network, learning rates of $p \in \{3 \times 10^{-4}, 1 \times 10^{-3}, 3 \times 10^{-3}, 1 \times 10^{-2}, 3 \times 10^{-2}, 1 \times 10^{-1}, 3 \times 10^{-1}\}$ are used. 3×10^{-4} is used as the lowest value tested because it was the learning rate used from before—and dropout only slows down the training process. From there, the rate is increased by a factor of around 3. It is expected that the higher learning rate values will result in nonconvergence. If this is not the case, then this part of the experiment will be repeated with a higher upper bound.

4.1.2 Iterating over dropout probability

The second part of the experiment will use the architecture

```
INPUT_{32×32×1} ->
CONV(3, 1)_{filters=32} -> RELU -> POOL(2, 2) ->
CONV(3, 1)_{filters=64} -> RELU -> POOL(2, 2) ->
DROP(p) ->
FC_384 -> RELU ->
FC_192 -> RELU ->
FC_10
```

where $p \in \{0.00, 0.10, 0.25, 0.50, 0.75\}$.

4.2 Implementation

Both parts of the experiment will use `for` loops to iterate over learning rate (in the case of the first part) or `keep_prob` (in the second part) in order to specify the appropriate hyperparameter value in the `params` objects—similar to the batch normalization implementation. As mentioned earlier, this experiment will use the TensorFlow function `tf.nn.dropout`.[9]

[9]Note that the TensorFlow dropout function `tf.nn.dropout` takes the argument `keep_prob`, which is $1 - p$.

4.3 Results and comments

4.3.1 Iterating over learning rate

The best-performing learning rate is 3×10^{-3} (figure 3). Learning rates of 1×10^{-1} and above do not converge, while learning rates of 1×10^{-3} and below do not converge fast enough for the purposes of further experiments to be conducted in this paper. The final validation error of the best-performing rate is 1.18, and the validation accuracy was 0.59. These results are actually not as good as the model this paper started with, although it is not clear why this is the case.

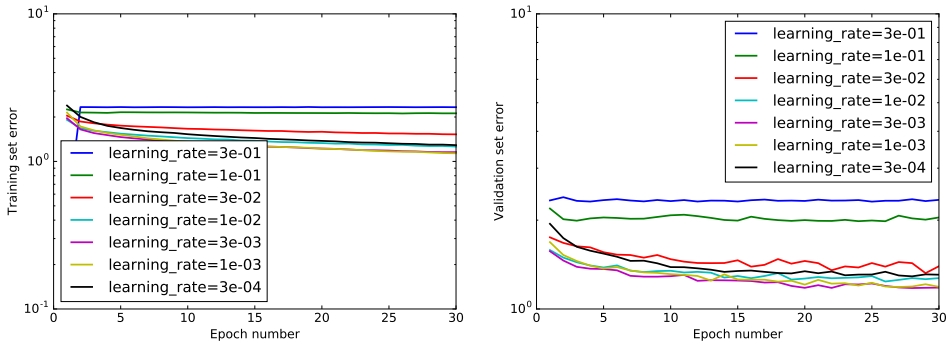

Figure 3: Training and validation error for different learning rates for multiple dropout layers.

4.3.2 Iterating over dropout probability

Since the last part of the experiment uses multiple dropout layers, the learning rates are not informative for this part. Therefore a few pilot runs were conducted, and it was decided to increase the learning rate by an order of magnitude from the batch normalization experiment from 10^{-4} to 10^{-3} in order to take the slower training time into account. The number of epochs was also changed to 15 since pilot runs showed this was enough time for most models to converge.

Dropout rates of $p = 0.50$ and $p = 0.25$ performed best in the 15-epoch period given, with validation error values of 0.96 and 0.95, respectively (figure 4). The validation accuracies for both were around 0.67. The effect on runtime was negligible.

The value $p = 0.25$ barely overtakes $p = 0.50$ in the last few epochs, when $p = 0.50$ appears to overfit a bit. It is unclear if the $p = 0.25$ validation error would have continued to decrease, but the shape of the curve suggests it may have required many more epochs in order to converge.

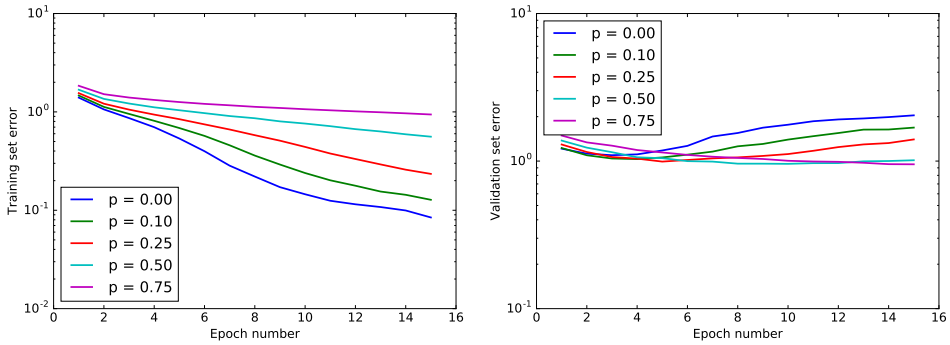

Figure 4: Training and validation error for different dropout rates.

4.3.3 Assessing both parts

The performance metrics in the second part of the experiment (where p was varied) are considerably better than those of the first part of the experiment. This was not expected, and it is unclear why this is the case. One possible explanation is that increasing the learning rate does not offset the slowdown in training time in a way that preserves performance—and therefore this degradation in performance is more severe when there are multiple dropout layers. Since multiple layers was presumed optimal by the original authors of the dropout paper, the results in this paper may be symptomatic of the particular network used in this paper—or vice versa.

4.4 Model changes

Going forward, a single dropout layer with $p = 0.50$ will be used instead of the more complete dropout model since it (1) is simpler; (2) converges more quickly; and (3) gives better validation error.[10] It is preferred against a dropout value of $p = 0.50$ because it converges a bit more quickly. The learning rate will also be increased to 10^{-3} to take the slower training into account.

5 Convolutional layers: kernel and stride sizes

The kernel size and stride size of convolutional layers have not had much variety in state-of-the-art image recognition tasks in the past few years [5, 13, 14, 4]. As mentioned in the section describing the design of the CNN, kernel width/height is often between 3 and 5, and the stride is often 1 or 2.

There is no well-established theory that explains why one kernel size would be better than another. However, due to the smaller number of computations, smaller kernels are at the very least expected

[10] However, this layer will not be included in architecture diagrams for the sake of concision.

to have shorter run times. Large strides are likely to have a decrease in performance since they result in smaller filters and therefore less information to carry forward in the network. But large strides carry the benefit of less computation and therefore shorter run time, since the kernel is applied a smaller number of times to each image.

This experiment will test the sensitivity of model performance and behavior to tweaks in these convolutional layer hyperparameters. Surely top researchers have done the same, but few have laid bare the tuning process, showing the effects of the tweaks on their results. This experiment will also make sure to observe whether the effects are stronger on the first convolutional layer or the second.

5.1 Description of experiment

For the first convolutional layer, kernel width values in $\{3, 5\}$ and stride values in $\{1, 2\}$ will be tested.[11] Similarly, for the second convolutional layer, the experiment will test kernel width values in $\{3, 5\}$ and stride values in $\{1, 2\}$. Taking all combinations of the four hyperparameters—each with two possible values—gives 16 combinations to be tested.

5.2 Implementation

A vector of all combinations of the hyperparameter values of the convolutional layers are expressed as the 16-element list `combos`, where each element is a 4-element list of hyperparameter values. A `for` loop is then used to assign these values to the `params` dictionary, which is then passed into the `run_cnn_model` function that runs the model.

```
...
params = {
    ...  # other hyperparameters
    'conv1_filter_size': None,  # to be assigned in loop
    'conv1_stride': None,       # to be assigned in loop
    'conv2_filter_size': None,  # to be assigned in loop
    'conv2_stride': None,       # to be assigned in loop
    ... # other hyperparameters
}
...
combos = [[3, 1, 3, 1], [3, 1, 3, 2], [3, 1, 5, 1], ..., [5, 2, 5, 2]]  # each
    element: [conv1_filter_size, conv1_stride, conv2_filter_size, conv2_stride]

for combo in combos:
    ...
    params['conv1_filter_size'] = combo[0]
    params['conv1_stride'] = combo[1]
    params['conv2_filter_size'] = combo[2]
    params['conv2_stride'] = combo[3]

    run_cnn_model(params)
    ...
```

[11] Kernel height is set to be the same as the width, giving square kernels.

. . .

5.3 Results and comments

5.3.1 Stride size

The best predictor of performance was stride size. In fact, the best-performing model had a stride size of 1 in both convolutional layers, while the worst-performing model had a stride size of 2. Their validation errors were 0.96 and 1.23, respectively, while their accuracies were 0.68 and 0.56. See figure 5, where these two models are compared and all other models are represented as light gray lines for comparison.

The fact that both the best- and worst-performing models had a kernel width of 3 means that small kernel sizes are especially sensitive to stride length. This is likely because small kernels perform well in this classification task—but when stride length is large, there is less relative overlap between kernel windows than when the kernel width is 5. This means that pixel patterns are more likely to be "split" by the kernels during the convolution process and therefore skipped over.[12]

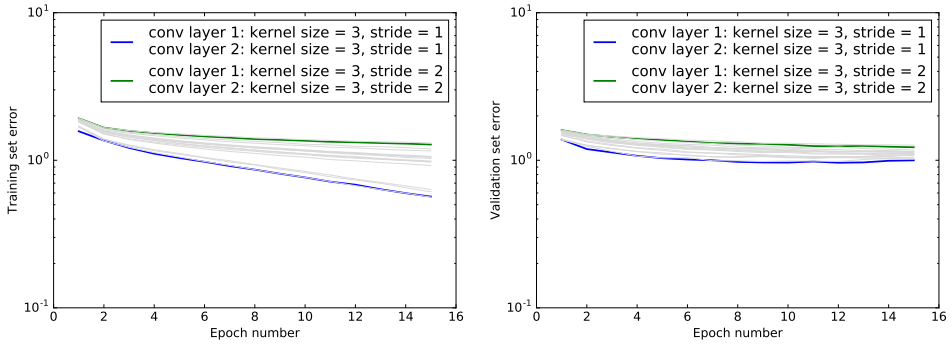

Figure 5: Effect of increasing stride when kernel width is 3.

However, one consolation of using a larger stride is much improved computation time, from 64 minutes to 9 (an 85% reduction) to train and validate the 15 epochs when the kernel width is kept at 3.

When kernel width is set to 3, increasing stride has a bit more of an effect when applied to the first layer as opposed to the second. This makes some sense, since some pixel patterns may be missed right at the beginning of the network, and this lack of information may propagate throughout the rest of the network.

[12] In other words, when the kernel width is 5 and the stride size is 1, overlap is 80%. When stride increases to 2, overlap decreases to 60%. On the other hand, when kernel width is 3, an increase in stride from 1 to 2 results in a decrease of overlap from 67% to just 33%—which is not only a larger jump in percentage, but a much lower level value when compared to the previous case where kernel width was equal to 5.

5.3.2 Kernel size

Increasing the kernel size resulted in a moderate increase in validation error, from 0.96 to 1.03 when stride was kept fixed at 1. However, the opposite was true when stride was set to 2, with error decreasing from 1.22 to 1.17. This again is likely due to improved overlap between filters during the convolution process.

Like stride, kernel size also has a large effect on the run time. When then kernel width is increased to 5 in both convolutional layers when keeping stride at 1, the run time increases by 79%—from 64 minutes to 115 minutes for all 15 epochs. When stride is kept at 2, there is a 60% increase in run time.

5.4 Model changes

No model changes will be made since the model with a kernel width of 3 and stride size of 1 in both convolutional layers has the best performance and a comparatively low run time when compared to the other models.

6 Pooling layers: width and stride sizes

The primary function of pooling layers is to decrease the size of the spatial dimensions in the network, and therefore reduce the number of parameters and amount of computation. The most common size of the pooling window is 2×2 with a stride length of 2, but other sizes are also used [4, 2].

This experiment will test the sensitivity of model performance and behavior to tweaks in these pooling layer hyperparameters. Again, top researchers have certainly done the same, but this experiment will dive a bit more deeply into why certain configurations may be preferable to others.

6.1 Description of experiment

This experiment design will more or less replicate that of the experiment that tweaked the convolutional layer hyperparameters. However, this experiment was a bit more targeted in that only 6 hand-selected combinations were tested instead of the possible 16.

6.2 Implementation

Similar to the last experiment, a vector of all combinations of the hyperparameter values of the pooling layers are expressed as the 6-element list `combos`, where each element is a 4-element list of

hyperparameter values. In this experiment, the `combos` list is

```
combos = [[2, 2, 2, 2], [3, 2, 3, 2], [3, 3, 3, 3], [3, 2, 2, 2], [2, 2, 3, 2], [2,
    1, 2, 1]]
```

where the first two elements of a list item are the width of the pooling window[13] and the stride of the *first* pooling layer, and the last two elements are the pooling window and the stride of the *second* pooling layer.

6.3 Results and comments

As with convolutional layers, the stride length is of enormous importance. When stride is increased from 1 to 2 in both pooling layers, the model converges much more slowly in terms of training steps (figure 6). However, improved run time due to the reduction in the number of parameters—and therefore computations—more than makes up for this. Specifically, when stride is equal to 1, the run time is 219 minutes—but when the stride is increased to 2, the run time decreases to 18% of the prior value, to 39 minutes. Furthermore, the minimum validation error of the model with a larger stride length is 0.959, which is actually a bit better than the 0.962 for the smaller stride length.[14]

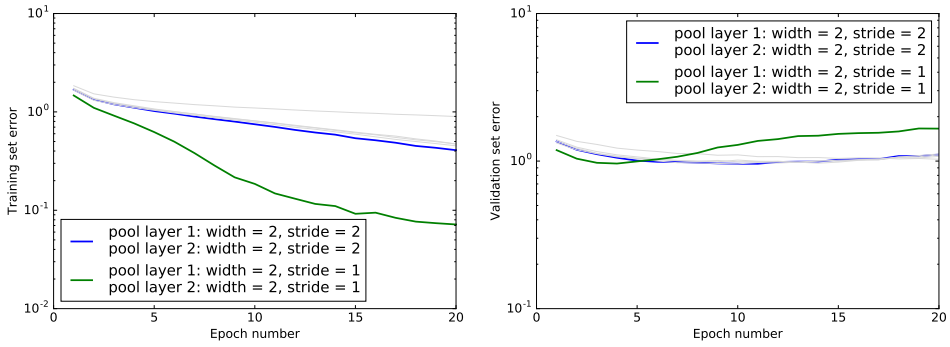

Figure 6: Effect of increasing stride in the pooling layer with a small window size.

Increasing window size, on the other hand, had only a slightly negative effect on performance, increasing validation error from 0.96 to 0.98. Window size also had little to no effect on run time, which is expected since window size does not affect the dimensionality of the data.

6.4 Model changes

Since a 2×2 pooling layer with a stride of 2 performs best, we will keep this configuration.

[13] As with the convolutional layers, the window size is assumed to be square, meaning the width is equal to the height.

[14] When the window size is kept fixed at 3, the same story holds, but to a lesser degree.

7 CIFAR-100

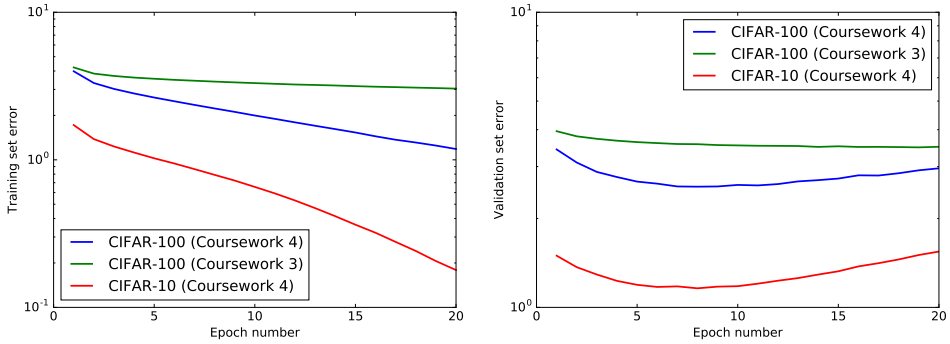

Figure 7: Training and validation error of CIFAR-100 models and the Coursework 4 CIFAR-10 model.

When the model up to this point is applied to the CIFAR-100 data, the performance is poorer than when it is used on the CIFAR-10 data. This is expected given that it is a more difficult classification problem. However, the performance is much better than the CIFAR-100 model from Coursework 3. The minimum validation error on the Coursework 4 CIFAR-100 model is 2.56, while on the Coursework 3 model it is 3.49. Perhaps more notably, the Coursework 4 validation accuracy of 0.37 is nearly double that of the Coursework 3 validation accuracy of 0.19 [1].

Figure 7 shows the error curves of the three networks. Note that there is serious overfitting with both Coursework 4 models, but this can be alleviated by early stopping or further regularization techniques.[15]

8 Summary

In the original **batch normalization** paper, the authors stated that batch normalization improved convergence speed in CNN models [8]. However, this did not take "wall time" into account [10]. When applied to the CIFAR-10 data using the CNN in this paper, batch normalization seemed to work best when applied to fully connected layers, improving convergence speed with no compromise to run time per epoch. However, when applied to convolutional layers, convergence suffered both in terms of training steps and run time per epoch. Overfitting did not seem to be prevented in any application of batch normalization to the network. These results counter the assertions made in the original paper.

In the original **dropout** paper, the authors recommended that dropout layers are spread throughout the network [12]. However, experiments on our network showed that this resulted in performance

[15] When it was not made explicit, the minimum validation error was taken and maximum accuracy

that was poorer than the original model, with slower convergence and lower validation error. Unexpectedly, a single dropout layer with $p = 0.5$ in the middle of the network with a boosted learning rate gave better validation error and accuracy, as well as faster convergence.

Many of the state-of-the-art CNN implementations use small **kernel sizes** and a small **stride length** in the **convolutional layers**, but the reasoning behind these choices are often not stated. This paper tweaked both hyperparameters to determine how they affect the behavior and performance of the model. Results showed that stride length is the most important parameter. When the kernel size is small, it is even more important to have a small stride length; otherwise, there is little overlap between the kernel windows, and the kernels may "skip" over patterns in the data. Differences in both kernel size and stride length resulted in dramatic differences in run time, with small kernel sizes and large strides resulting in faster training.

When tweaking the **pooling layer** parameters of **window size** and **stride length**, it was found that stride length was again the most important determinant of performance. A stride length of 2 resulted in better performance than a stride of 1—and it had a much shorter training time when taking wall time into account. Window size did not affect run time, which was expected.

When the model was applied to the CIFAR-100 data, accuracy was nearly double that of the Coursework 3 CIFAR-100 model.

Appendix

CIFAR-10 classes

Figure 8 shows example images from each of the 10 classes from the CIFAR-10 data set.

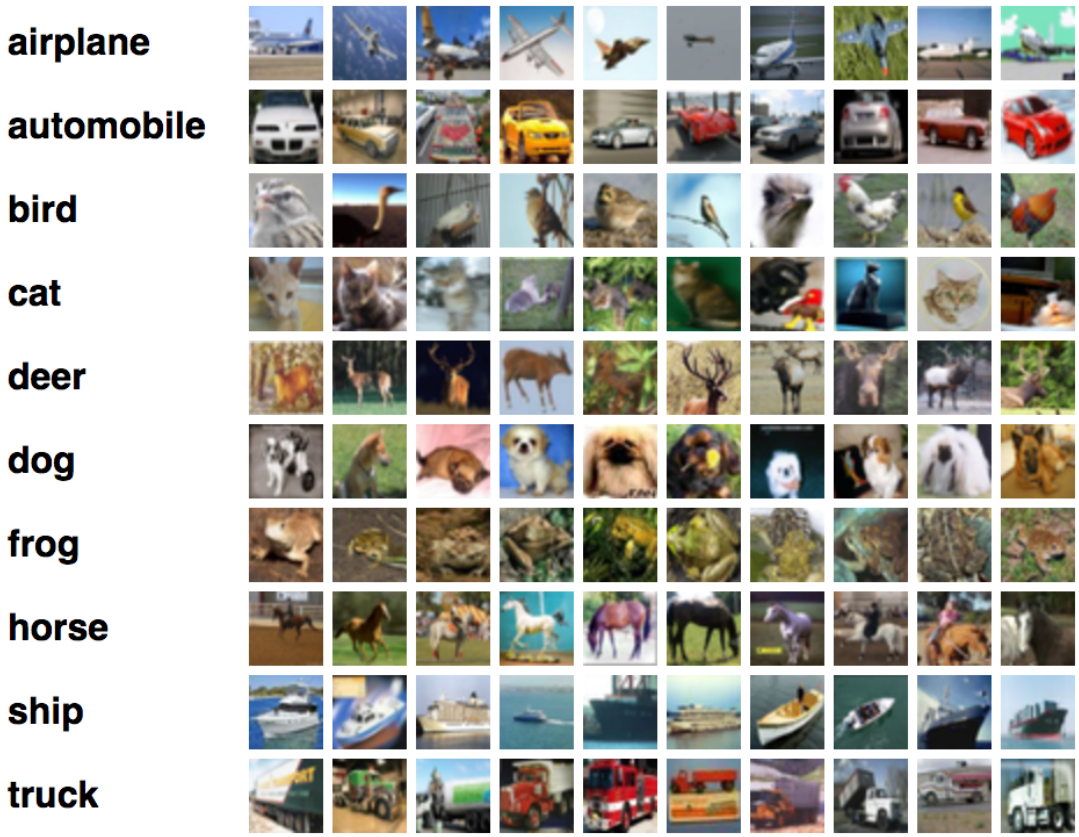

Figure 8: CIFAR-10 example images [3].

CIFAR-100 classes

Below are the 100 classes form the CIFAR-100 data set. In bold are the "superclasses" to which the classes belong. There are 5 classes in each superclass.

Superclass	Class labels
aquatic mammals	beaver, dolphin, otter, seal, whale
fish	aquarium fish, flatfish, ray, shark, trout
flowers	orchids, poppies, roses, sunflowers, tulips
food containers	bottles, bowls, cans, cups, plates
fruit and vegetables	apples, mushrooms, oranges, pears, sweet peppers
household electrical devices	clock, computer keyboard, lamp, telephone, television
household furniture	bed, chair, couch, table, wardrobe
insects	bee, beetle, butterfly, caterpillar, cockroach
large carnivores	bear, leopard, lion, tiger, wolf
large man-made outdoor things	bridge, castle, house, road, skyscraper
large natural outdoor scenes	cloud, forest, mountain, plain, sea
large omnivores and herbivores	camel, cattle, chimpanzee, elephant, kangaroo
medium-sized mammals	fox, porcupine, possum, raccoon, skunk
non-insect invertebrates	crab, lobster, snail, spider, worm
people	baby, boy, girl, man, woman
reptiles	crocodile, dinosaur, lizard, snake, turtle
small mammals	hamster, mouse, rabbit, shrew, squirrel
trees	maple, oak, palm, pine, willow
vehicles 1	bicycle, bus, motorcycle, pickup truck, train
vehicles 2	lawn-mower, rocket, streetcar, tank, tractor

CNN architecture design notes

Instructional examples

There seems to be a consistent pattern regarding the number of units and filters for each layer used in popular instructional implementations on the CIFAR and MNIST data.[16]

For exampe, TensorFlow's instructional implementation for MNIST is

$$
\begin{aligned}
&\texttt{INPUT}_{28\times 28\times 1} \texttt{ ->} \\
&\texttt{CONV(5, 1)}_{filters=32} \texttt{ -> RELU -> POOL(2, 2) ->} \\
&\texttt{CONV(5, 1)}_{filters=64} \texttt{ -> RELU -> POOL(2, 2) ->} \\
&\texttt{FC}_{1024} \texttt{ -> RELU ->} \\
&\texttt{FC}_{10}
\end{aligned}
$$

where subscripts indicate the output dimensionality of an image, `CONV(`w`,`s`)` indicates a convolutional layer with width (and height) w and stride size s, and `POOL(`w`,`s`)` indicates a pooling layer with width (and height) w and stride size s.

TensorFlow's instructional CIFAR-10 implementation follows a similar pattern[17]:

[16] MNIST is a data base of handwritten digits. Similar to CIFAR, it is a common task to have to classify the MNIST data [15].

[17] A few things are not captured in the architecture notation. For example, batch normalization was applied after

```
INPUT_{24×24×1} ->
CONV(5, 1)_{filters=64} -> RELU -> POOL(3, 2) ->
CONV(5, 1)_{filters=64} -> RELU -> POOL(3, 2) ->
FC_{384} -> RELU ->
FC_{192} -> RELU ->
FC_{10}
```

An implementation from a popular instructional video for classifying the CIFAR-10 data is below[18] [16]:

```
INPUT_{24×24×1} ->
CONV(5, 1)_{filters=64} -> RELU -> POOL(2, 2) ->
CONV(5, 1)_{filters=64} -> RELU -> POOL(2, 2) ->
FC_{256} -> RELU ->
FC_{128} -> RELU ->
FC_{10}
```

In the examples above, it was common to use the Adam optimizer with a learning rate between 10^{-4} and 10^{-3}. For weight initialization, it was common to use TensorFlow's `tf.truncated_normal` and `tf.random_normal` functions. Bias was initialized using with `tf.random_normal` or `tf.constant` with either zero or a slightly positive value like 0.1. The error function was multi-class cross entropy error with softmax.

State-of-the-art examples

CNN models that have the highest performance on the CIFAR, MNIST and other common baseline machine learning data sets are often much more complex than the instructional examples described above. Examples include models that won or performed well in the ImageNet Large Scale Visual Recognition Challenge (ILSVRC),[19] such as GoogLeNet, VGGNet and ResNet.

GoogLeNet won the ILSVRC in 2014. It is known for its so-called *Inception* model. It differs from the instructional models above in that it is deeper (hence the name [17]), claiming the model uses 22 layers—although this understates the complexity. It also seems that some layer types are not counted. The kernel sizes used in the convolutional layers are varied, including 1×1,[20] 3×3 and 5×5. The model uses average instead of fully connected layers as the final layers of the network [14, 4].

The runner-up to the ILSVRC in 2014 was VGGNet. The VGGNet was appealing in its simplicity: It consisted almost completely of convolutional layers with 3×3 convolutions and pooling layers

the first pooling layer and *before* the second pooling layer. Data augmentation was also performed on the input data. A careful observer may notice that the inputs here are of size 24×24 when the CIFAR-10 images are of size 32×32; this is because the images were cropped as part of the data augmentation process.

[18] Again, there was also data augmentation and batch normalization.

[19] For more information, see `http://www.image-net.org/challenges/LSVRC/`.

[20] One may ask why convolutions of kernel size 1×1 are used. The answer is that they provide dimensionality reduction, since they still span all of the channels of the tensors.

with window size 2×2. The best-performing models explained in the follow-up paper had 16–19 layers. The number of filters in the convolutional layers that were tested were 64, 128, 256, and 512, while the fully connected layers had 4096 units [5, 4].

ResNet won the ILSVRC in 2015. Its architecture was quite different from prior winners, making use of skip connections and strong batch normalization. It also uses orders of magnitude more layers than previous winners: On the ImageNet data set, it used 152, while on CIFAR-10, up to 1000 layers were tested. The architecture also does not have fully connected layers as the final layers [13, 4].

Batch normalization function

```
def batch_normalization(layer):

    '''
    Perform batch normalization on a layer. The layer argument can be a 4-D tensor
    or a flattened layer.

    Input:
        layer: TensorFlow layer object
    Output:
        batch-normalized layer
    '''

    # Save original shape of layer and determine if it has the dimensionality of an
    image.
    orig_layer_shape = [-1] + layer.get_shape().as_list()[1:]
    is_image = len(orig_layer_shape) == 4

    if is_image:
        # Use separate function for 4-D tensors.
        return tf.nn.lrn(layer)

    dim = np.prod(orig_layer_shape[1:])  # Get flattened dimension
    epsilon = 1e-6
    batch_mean1, batch_var1 = tf.nn.moments(layer, [0])
    z1_hat = (layer - batch_mean1) / tf.sqrt(batch_var1 + epsilon)
    scale1 = tf.Variable(tf.ones([dim]))
    beta1 = tf.Variable(tf.zeros([dim]))
    layer_norm = tf.nn.batch_normalization(layer, batch_mean1, batch_var1, beta1,
    scale1, epsilon)

    return layer_norm
```

References

[1] s1667278, "Machine learning practical: Coursework 3." University of Edinburgh course assignment, Feb. 2017.

[2] TensorFlow, "Convolutional neural networks." Website available at https://www.tensorflow.org/tutorials/deep_cnn. Accessed 5 Mar 2017.

[3] A. Krizhevsky, "The cifar-10 dataset." Website available at https://www.cs.toronto.edu/ kriz/-cifar.html. Accessed 9 Mar 2017.

[4] J. J. Fei-Fei Li, Andrej Karpathy, "Cs231n: Convolutional neural networks for visual recognition." Stanford University course. Website available at http://cs231n.stanford.edu/. Accessed 13 Mar 2016.

[5] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *CoRR*, vol. abs/1409.1556, 2014.

[6] TensorFlow, "Deep mnist for experts." Website available at https://www.tensorflow.org/get_-started/mnist/pros. Accessed 5 Mar 2017.

[7] A. Damien, "convolutional_network." Website available at https://github.com/aymericdamien/TensorFlow-Examples/blob/master/notebooks/3_Neural-Networks/convolutional_network.ipynb. Accessed 5 Mar 2017.

[8] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," *CoRR*, vol. abs/1502.03167, 2015.

[9] s1667278, "Machine learning practical: Coursework 2." University of Edinburgh course assignment, Nov. 2016.

[10] F. Schilling, "The effect of batch normalization on deep convolutional neural networks," 2016.

[11] S. Renals, "Machine learning practical." University of Edinburgh course. Website available at http://www.inf.ed.ac.uk/teaching/courses/mlp/2016. Accessed 13 Mar 2016.

[12] N. Srivastava, G. E. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting.," *Journal of Machine Learning Research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[13] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016.

[14] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2015.

[15] Y. LeCun, "The mnist database of handwritten digits." Website available at http://yann.lecun.com/exdb/mnist/. Accessed 5 Mar 2017.

[16] M. E. H. Pedersen, "06_cifar-10." Website available at https://github.com/Hvass-Labs/TensorFlow-Tutorials/blob/master/06_CIFAR-10.ipynb. Accessed 5 Mar 2017.

[17] "Know your meme: We need to go deeper." Website available at http://knowyourmeme.com/memes/we-need-to-go-deeper. Accessed 15 Sep 2014.