Image Classification: CNN

Final Lab Part 2

Carol Rameder - 14021218
Hamed Ahadi - 14109379
Lisa van Ommen - 14657651

Computer Vision

22/10/2022

Table of Contents

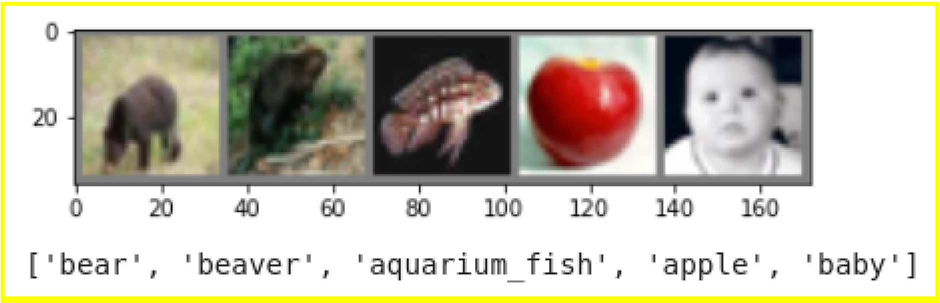
1. CIFAR-100

The goal of this report is to present and explain image classification using CNNs (Convolutional Neural Networks).

1.1 Installation

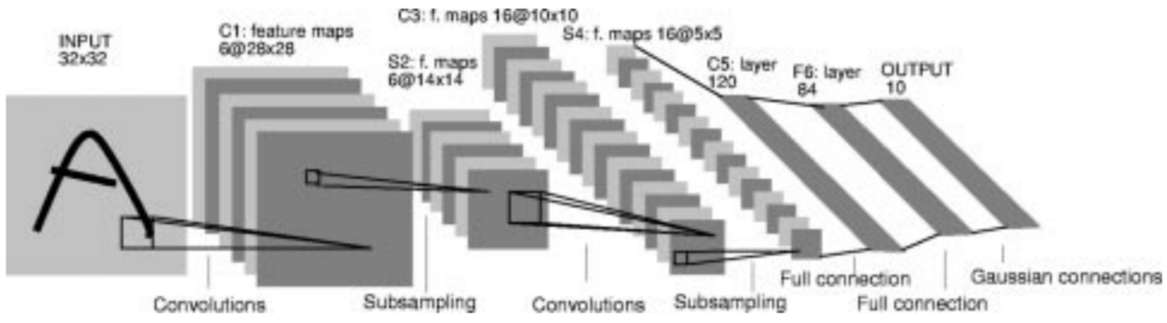
This classification will be done using the framework PyTorch and the dataset CIFAR-100. PyTorch is a machine learning framework that provides both Tensor computation with strong GPU acceleration and deep neural networks. This is beneficial in the case of image classification since that generally is computationally expensive.

The data used to perform the classification is from the CIFAR-100 dataset, which contains 20 superclasses and 100 fine-grained classes, containing 500 training images and 100 testing images each, resulting in a total of 50.000 training images and 10.000 images. An image example for the first five classes (with labels from 0 to 4) and their name can be seen in the picture below.



```
['bear', 'beaver', 'aquarium_fish', 'apple', 'baby']
```

1.2 Architecture understanding



The classification can be done with two different architectures, one of them is an ordinary two-layer network, and the other is a Convolutional Neural Network (CNN), using the structure of LeNet-5 (Lecun et al.) shown in the scheme above. The two-layer network has fully connected layers and a ReLU activation function, making it structure-agnostic and broadly applicable (Mahajan). Contrary to the two-layer network, in the Convolutional Network, not every neuron is connected. Instead, it uses local connectivity -Convolution layers- and then shares weight parameters, which makes it much faster in training and very suitable for large amounts of data. In the Convolution layers, a kernel matrix is convoluted over the input to extract relevant features and then the result is subsampled using the AvgMaxPool function. Both of these networks need to be fed colour images represented in multidimensional tensors. After the feed-forward process, the output class of the network is compared with the gold label and summed with a lossfunction.

In the case of the Convolutional Network, the first convolutional layers have a kernel size of *5*. The F6-layer has *10164* trainable parameters.

The structure of the Two-Layer Network

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Linear-1                  [-1, 128]         393,344
            Linear-2                  [-1, 100]          12,900
================================================================
Total params: 406,244
Trainable params: 406,244
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.01
Forward/backward pass size (MB): 0.00
Params size (MB): 1.55
Estimated Total Size (MB): 1.56
----------------------------------------------------------------
```

The structure of the Convolutional Neural Network

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1            [-1, 6, 28, 28]             456
         MaxPool2d-2            [-1, 6, 14, 14]               0
            Conv2d-3           [-1, 16, 10, 10]           2,416
         MaxPool2d-4             [-1, 16, 5, 5]               0
            Conv2d-5            [-1, 120, 1, 1]          48,120
            Linear-6                   [-1, 84]          10,164
            Linear-7                  [-1, 100]           8,500
================================================================
Total params: 69,656
Trainable params: 69,656
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.01
Forward/backward pass size (MB): 0.06
Params size (MB): 0.27
Estimated Total Size (MB): 0.34
----------------------------------------------------------------
```

1.3 Preparation of training

Before training the models, it is important to prepare the training and testing data. The CIFAR100_loader class creates an interface that makes the transformation process of the data, from the folder to the model, easily accessible. The Dataset is responsible for accessing and processing single instances of data. The DataLoader pulls instances of data from the Dataset, collects them in batches, and returns them for consumption by your training loop.

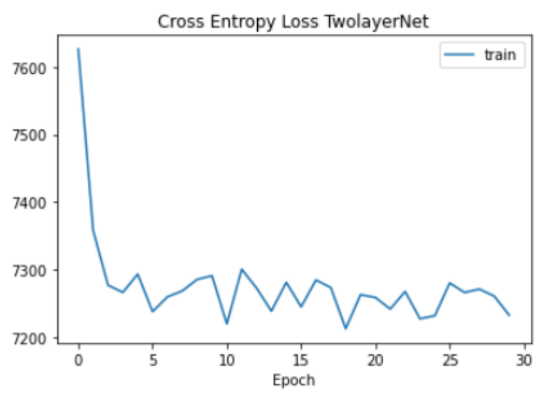
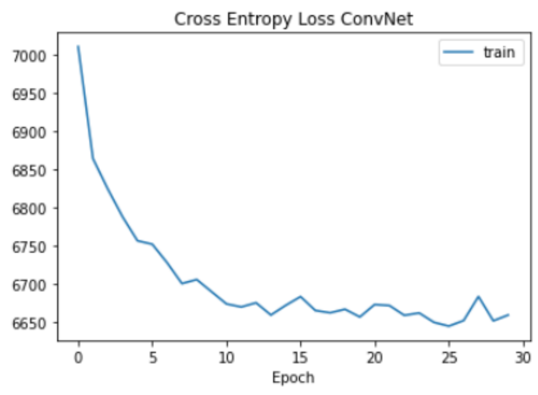
The goal of this process is to modify the data in a format such that the neural networks are able to perform the learning process (feedforward and backpropagation). In the input file, the RGB channels of a single image were flattened and stored sequentially on a single line. Therefore, each line stored 3072 integer values, 1024 for each of the three colour channels, which has been reshaped in the standard matrix format of 32x32x3.

In the last step, the data has been transformed with a specific pipeline and brought to a tensor format to make it ready to feed into the model. Thus, the images were normalized to a smaller range than the standard RGB, and random vertical and horizontal flips were performed. The purpose of this is to make the network robust to these changes that can appear in real-life situations and increase the chances to learn the same features aligned differently from one case to another.

Finally, the images are loaded into separate testing and training sets and the training set is split into iterable batches of pre-set size for the learning process.

Generally, the model learns how to categorize a new data point from the test set with the features learned from processing the training data. During the training process, all the images from the same batch are forward propagated into the net producing an output. According to these predicted values, the error of the model, with the respect to the true labels, is computed with a pre-set loss function. In the backpropagation phase, the model parameters are updated trying to minimize the value of the loss function. Thus, after processing the whole batch, the gradients are computed and the weights are updated accordingly, moving the model closer to the global optimum. This process is called Batch Learning and in most cases, it converges faster than processing each instance individually.

The procedure to update the weights according to their contribution to the error is performed with an optimizer. Considering the trainable parameters of the model the input in the multidimensional space of the loss function space, the direction in which the model should go (or how the weights are updated) is computed with the optimizer. In our case, the Adam algorithm was used as it is generally known to have a great performance in common deep learning and computer vision problems when using a CNN (Nanni et al. #) and in comparison with other common optimizers such as RMSProp, Stochastic Gradient Descent Nesterov or AdaDelta (Gugger). The loss function chosen was Cross Entropy Loss.

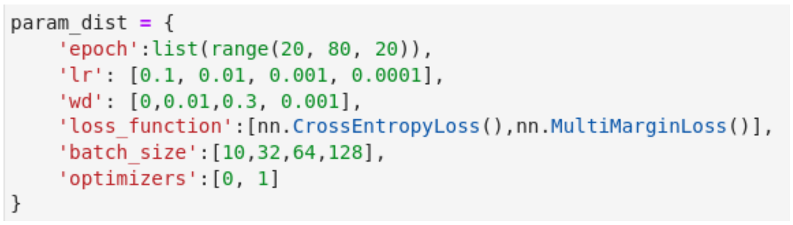
The performance of the Two-Layer net during the epochs	The performance of the CNN during the epochs
	

Accuracy was the performance measure selected for our experiment. It is averaged over the figures of all 100 fine-grained classes of CIFAR-100. Both networks recorded 8% accuracy on the test set. The evolution of the loss function figures over the epochs is more constant for the CNN.

1.4.1 Hyperparameters

The performance of the vast majority of Machine learning (ML) algorithms such as gradient boosting, random forest and neural networks for regression and classification is highly dependent on the hyperparameters of these algorithms. They have to be set before running them. In contrast to direct, first-level model parameters, which are determined during training, these second-level tuning parameters often have to be carefully optimized to achieve maximal performance (Hutter et al. #). Even though most algorithm implementations and the available Python libraries have pre-set default values for these parameters their accuracy may be mediocre if they are not adapted. Thus, there is a need to find the best possible combination of these parameters, given the time and data available.

A simple but efficient tuning strategy is Random Search. Using this method, we define distributions for each hyperparameter with possible values to be selected during the validation phase. Compared to similar methods such as Grid Search, in Random Search not all the values are tested. For each iteration, a configuration of all hyperparameters is selected from the pre-defined distribution with possible. The algorithm is run and its accuracy is compared with the others. If the current set of values determined the best local accuracy, this configuration is considered to be the closest to the global and kept for further experiments.



```
param_dist = {
    'epoch':list(range(20, 80, 20)),
    'lr': [0.1, 0.01, 0.001, 0.0001],
    'wd': [0,0.01,0.3, 0.001],
    'loss_function':[nn.CrossEntropyLoss(),nn.MultiMarginLoss()],
    'batch_size':[10,32,64,128],
    'optimizers':[0, 1]
}
```

The implemented hyperparameter search space in our experiment can be seen in the picture above. The pre-set possible values are experimental. The number of iterations was set to 25. The considered parameters are:

- Batch size: namely how many images are to be processed at each step. It determines the number of iterations required to process all images and proceed to the next epoch.
- Learning rate (‘lr’): how much the weights of the network will be updated at each step of the optimizer. It determines how fast the model learns by how much the weights can be changed. Usually, a small learning rate comes with the need for a larger number of epochs. A possible improvement for further research would be an adaptive learning rate.
- Epochs: how many iterations the network will run until convergence. It determines how many times each data point is fed through the model.
- Loss function: how the loss of the network is computed. Next to the initially used CrossEntropy, Multi MarginLoss is added
- Optimizers: Adam (‘0’) or Adadelta (‘1’). There are certain cases in which Adam is not able to converge to the global optimal solution (Reddi et al. #). Therefore, there is a need to try other optimizers and not only change the values of the other hyperparameters. They were compatible with the experiment as well, as both accept specified learning rate and weight decay
- Weight decay (‘wd’): also known as L2 regularization, it is a regularization technique that is used to penalize the size of the weights of a model by adding a factor to the loss coefficient. It helps to prevent overfitting (Kumar)

The normalization values in the transform function applied for both training and testing data were modified compared with the initial ones from mean = 0.5 and standard deviation of 0.5 for all 3 channels to values adapted to the CIFAR-100 database: mean = (0.507, 0.486, 0.440) and standard deviation of (0.267, 0.256, 0.276) for the RGB channels accordingly. These were found online experimentally by other researchers.

After experimenting with these hyperparameters for both of the networks, the highest accuracy obtained for the two-layer network was 10% and the best accuracy convolutional network for the Convolutional Network is 12%. It was reached with the hyperparameters set to the values as can be seen in table 1.

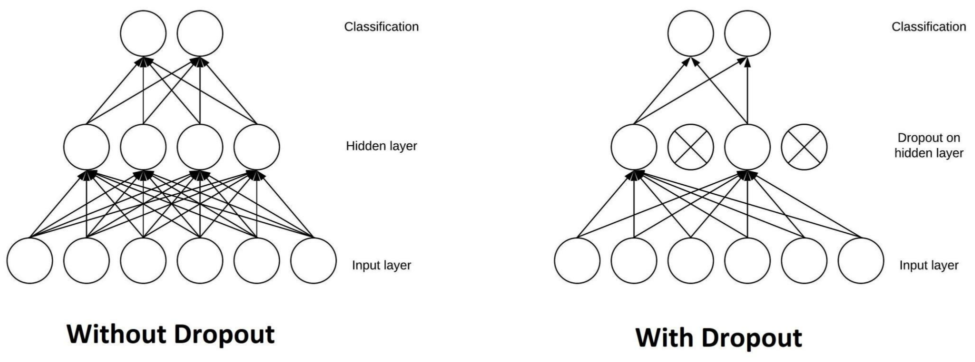
Hyperparameters (epoch, lr, wd, loss_f, batch_size, optimizer)	Two-layer network	CNN
Initial: 40, 0.001, 0.001, Cross_Entropy, 32, Adam	8%	8%
After Random Search: 30, 0.01, 0.001, Cross_Entropy, 64, Adam	10%	12%

Table 1. Settings of the hyperparameters.

1.4.2 Adding Layers

Besides changing the hyperparameters and modifying the functions, the architectures of the networks themselves are usually highly impactful for the behaviour and the performance of the models. In our experiment, two types of layers were added in certain positions to analyze if this really improves the performance of our models on the same dataset.

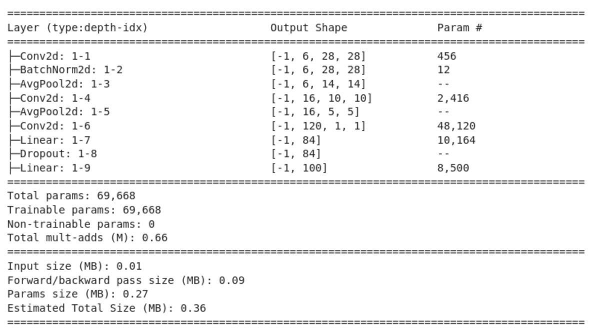
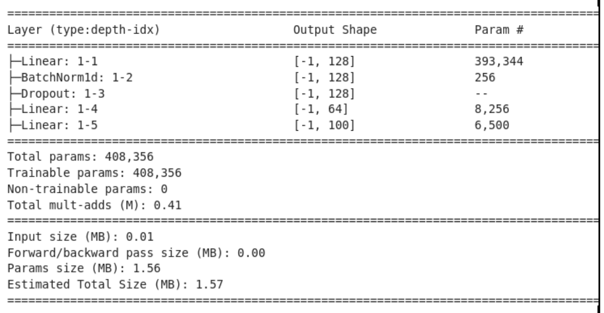
First of all, the Dropout allows our model to become weight-independent during the training process by randomly dropping weights in a chosen layer (Brilenkov). More precisely, during both front and backpropagation through the network, the nodes of a Dropout Layer are set inactive and therefore, the information does not flow through them. This helps to diminish the overtraining problem in powerful Deep Neural nets with a large number of parameters by making the network more robust and reducing its variance on supervised learning tasks such as vision recognition (Srivastava et al.).



Secondly, Batch Normalisation also reduces overfitting by preventing our NN from the exploding gradient issue. More precisely, the output values of some hidden nodes become very large and are propagated further and thus, they affect what previously has been learned. To solve this issue, the output values of a hidden layer are filtered with a Batch Normalization Layer. This is done by computing the mean and the standard deviation of the outputs of the previous layer and normalizing the values with the formula below and squeezing them in an interval centred in 0 and with a standard deviation equal to 1.

$$y = \frac{x - \mathrm{E}[x]}{\sqrt{\mathrm{Var}[x] + \epsilon}} * \gamma + \beta$$

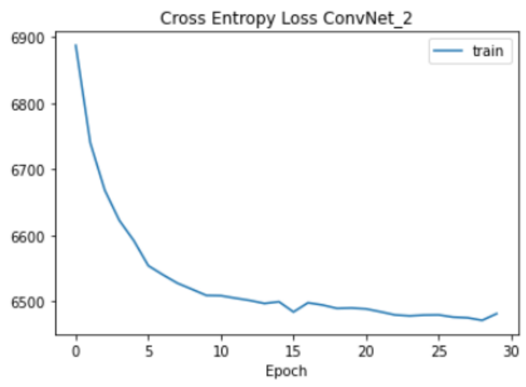
In our experiment, the Dropout and Batch Normalization layers were added to both models, the CNN and the Two Layer Network. Dropout was added before the last output layer of both NN as suggested in the article of Ruslan Brilenkov, whilst the Batch Normalization was added after the first layer, to prevent large values to be propagated in the next layers.

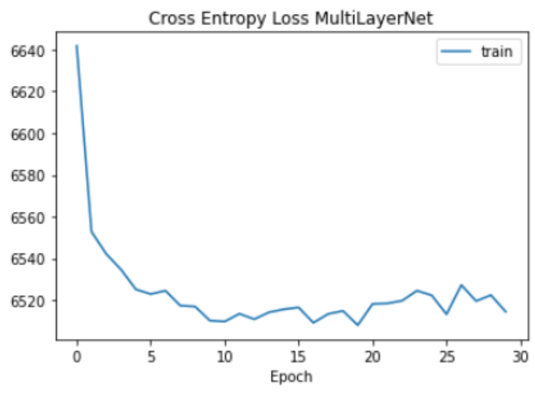
The Structure of the CNN after adding the layers	The Structure of the Multi-Layer Network
	

```
==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
├─Conv2d: 1-1                            [-1, 6, 28, 28]           456
├─BatchNorm2d: 1-2                       [-1, 6, 28, 28]           12
├─AvgPool2d: 1-3                         [-1, 6, 14, 14]           --
├─Conv2d: 1-4                            [-1, 16, 10, 10]          2,416
├─AvgPool2d: 1-5                         [-1, 16, 5, 5]            --
├─Conv2d: 1-6                            [-1, 120, 1, 1]           48,120
├─Linear: 1-7                            [-1, 84]                  10,164
├─Dropout: 1-8                           [-1, 84]                  --
├─Linear: 1-9                            [-1, 100]                 8,500
==========================================================================================
Total params: 69,668
Trainable params: 69,668
Non-trainable params: 0
Total mult-adds (M): 0.66
==========================================================================================
Input size (MB): 0.01
Forward/backward pass size (MB): 0.09
Params size (MB): 0.27
Estimated Total Size (MB): 0.36
==========================================================================================
```

```
==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
├─Linear: 1-1                            [-1, 128]                 393,344
├─BatchNorm1d: 1-2                       [-1, 128]                 256
├─Dropout: 1-3                           [-1, 128]                 --
├─Linear: 1-4                            [-1, 64]                  8,256
├─Linear: 1-5                            [-1, 100]                 6,500
==========================================================================================
Total params: 408,356
Trainable params: 408,356
Non-trainable params: 0
Total mult-adds (M): 0.41
==========================================================================================
Input size (MB): 0.01
Forward/backward pass size (MB): 0.00
Params size (MB): 1.56
Estimated Total Size (MB): 1.57
==========================================================================================
```

The performance of both models has dropped after adding the mentioned layers from 8% to 7%. In both cases, the most likely explanation is the low size and resolution of the images and the increased number of classes to be predicted. Numerous classes are correctly predicted with 0% accuracy, compared to others, which are correctly classified in more than 40% of the cases. To put it simply, undertraining caused both NN not to be able to escape the local minimum. However, as observed in the figures below, the models do decrease the loss function, thus we can affirm that the learning process works.

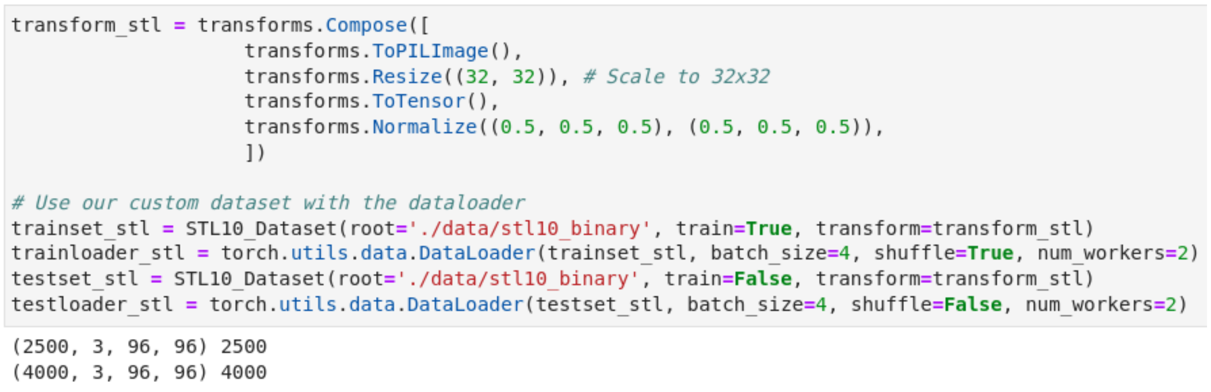




There are quite some differences between these two networks in different aspects: architecture, performance and learning rates. Chapter 1.2 touched on the more theoretical side of these differences, but they can also be seen in practice. Even though ConvNet_2 contains Convolution layers, which should have brought a significant improvement in performance compared to a rather large Multi-Layer Net, especially after adding another hidden layer with 64 hidden nodes which makes it prone to overfitting.

2. STL-10 Dataset

As observed in the image below, loading the STL-10 Dataset works successfully. The full implementation of the STL10_Dataset can be seen in the notebook file attached.



```
transform_stl = transforms.Compose([
                    transforms.ToPILImage(),
                    transforms.Resize((32, 32)), # Scale to 32x32
                    transforms.ToTensor(),
                    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
                    ])

# Use our custom dataset with the dataloader
trainset_stl = STL10_Dataset(root='./data/stl10_binary', train=True, transform=transform_stl)
trainloader_stl = torch.utils.data.DataLoader(trainset_stl, batch_size=4, shuffle=True, num_workers=2)
testset_stl = STL10_Dataset(root='./data/stl10_binary', train=False, transform=transform_stl)
testloader_stl = torch.utils.data.DataLoader(testset_stl, batch_size=4, shuffle=False, num_workers=2)
```

```
(2500, 3, 96, 96) 2500
(4000, 3, 96, 96) 4000
```

Bibliography

Brilenkov, Ruslan. “2 Layers to Greatly Improve Keras CNN | by Ruslan Brilenkov.” *DataDrivenInvestor*, 20 October 2021, https://medium.datadriveninvestor.com/2-layers-to-greatly-improve-keras-cnn-1d4d1c3e8ea5. Accessed 22 October 2022.

Gugger, Sylvain. “fast.ai - AdamW and Super-convergence is now the fastest way to train neural nets.” *Fast.ai*, 2 July 2018, https://www.fast.ai/posts/2018-07-02-adam-weight-decay.html. Accessed 22 October 2022.

“How ReLU and Dropout Layers Work in CNNs.” *Baeldung*, 12 May 2022, https://www.baeldung.com/cs/ml-relu-dropout-layers. Accessed 22 October 2022.

Hutter, Frank, et al. “An efficient approach for assessing hyperparameter importance.” *Proceedings of the 31st International Conference on Machine Learning*, 2014, http://proceedings.mlr.press/v32/hutter14.html.

Ismiguzel, Idil. “Hyperparameter Tuning with Grid Search and Random Search.” *Towards Data Science*, 2021, https://towardsdatascience.com/hyperparameter-tuning-with-grid-search-and-random-search-6e1b5e175144.

Kumar, Ajitesh. “Weight Decay in Machine Learning: Concepts.” *Data Analytics*, 7 June 2022, https://vitalflux.com/weight-decay-in-machine-learning-concepts/. Accessed 22 October 2022.

Lecun, Y., et al. “Gradient-based learning applied to document recognition.” *IEEE Xplore*, https://ieeexplore.ieee.org/abstract/document/726791.

Mahajan, Pooja. “Fully Connected vs Convolutional Neural Networks | by Pooja Mahajan | The Startup.” *Medium*, 22 October 2020, https://medium.com/swlh/fully-connected-vs-convolutional-neural-networks-813ca7bc6ee5. Accessed 22 October 2022.

Nanni, Loris, et al. “"Exploiting Adam-like Optimization Algorithms to Improve the Performance of Convolutional Neural Networks."” 2021, https://arxiv.org/abs/2103.14689.

Reddi, Sashank, et al. “On the convergence of adam and beyond.” 2019, https://arxiv.org/abs/1904.09237.

Srivastava, Nitish, et al. “Dropout: A Simple Way to Prevent Neural Networks from Overfitting.” University of Toronto - Department of Computer Science, 2014, https://www.jmlr.org/papers/volume15/srivastava14a/srivastava14a.pdf?utm_content=buffer79b43&utm_medium=social&utm_source=twitter.com&utm_campaign=buffer,.