1. Is it OK to initialize all the weights to the same value as long as that value is selected randomly using He initialization?

Name three advantages of the SELU activation function over ReLU.

In which cases would you want to use each of the following activation functions: SELU, leaky

ReLU (and its variants), ReLU, tanh, logistic, and softmax? 5. What may happen if you set the momentum hyperparameter too close to 1 (e.g., 0.99999) when using an SGD optimizer? 6. Name three ways you can produce a sparse model. 7. Does dropout slow down training? Does it slow down inference (i.e., making predictions on new instances)? What about MC Dropout? 8. Practice training a deep neural network on the CIFAR10 image dataset: a. Build a DNN with 20 hidden layers of 100 neurons each (that’s too many, but it’s the point of this exercise). Use He initialization and the ELU activation function. b. Using Nadam optimization and early stopping, train the network on the CIFAR10 dataset. You can load it with keras.datasets.cifar10.load\_​data(). The dataset is composed of 60,000 32 × 32–pixel color images (50,000 for training, 10,000 for testing) with 10 classes, so you’ll need a softmax output layer with 10 neurons. Remember to search for the right learning rate each time you change the model’s architecture or hyperparameters. c. Now try adding Batch Normalization and compare the learning curves: Is it converging faster than before? Does it produce a better model? How does it affect training speed? d. Try replacing Batch Normalization with SELU, and make the necessary adjustements to ensure the network self-normalizes (i.e., standardize the input features, use LeCun normal initialization, make sure the DNN contains only a sequence of dense layers, etc.). e. Try regularizing the model with alpha dropout. Then, without retraining your model, see if you can achieve better accuracy using MC Dropout.

1. Is it OK to initialize the bias terms to 0?

Yes, it is generally acceptable to initialize the bias terms to 0. However, it is important to note that other values may result in better performance, depending on the type of model and data.

1. Name three advantages of the SELU activation function over ReLU.

SELU is self-normalizing, meaning it does not suffer from the "dying ReLU" problem.

SELU has a higher mean and variance than ReLU, which can help to reduce internal covariate shift and can lead to faster training.

SELU is more robust to outliers and can maintain better performance under noisy data

1. In which cases would you want to use each of the following activation functions: SELU, leaky ReLU (and its variants), ReLU, tanh, logistic, and softmax?

SELU: When training deep neural networks and when dealing with noisy data or outliers.

Leaky ReLU (and its variants): When training deep neural networks and when dealing with data that is not linearly separable.

ReLU: When training deep neural networks and when dealing with data that is linearly separable.

Tanh: When training shallow neural networks and when dealing with data that is non-linear.

Logistic: When training shallow neural networks and when dealing with binary classification tasks.

Softmax: When training shallow neural networks and when dealing with multi-class classification tasks.

1. What may happen if you set the momentum hyperparameter too close to 1 (e.g., 0.99999) when using an SGD optimizer?

If the momentum hyperparameter is set too close to 1, the SGD optimizer may cause oscillations in the optimization trajectory and can lead to slow convergence or even divergence of the optimization process. This is because the SGD optimizer will attempt to move too quickly in the direction of the previous update, resulting in overshooting the optimum.

1. Name three ways you can produce a sparse model.

Use L1 regularization, which adds a penalty on the sum of the absolute values of the weights in the model. This encourages the model to reduce the number of non-zero weights, leading to a sparse model.

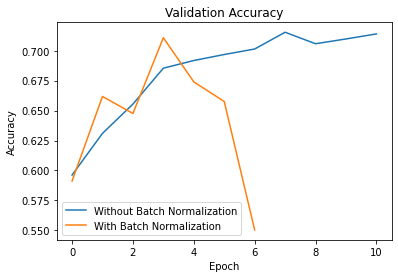
Use feature selection techniques such as forward selection, backward selection, or recursive feature elimination to select only the most relevant features in the model. This can reduce the number of inputs, leading to a sparse model.

Use pruning techniques such as magnitude pruning or low-rank factorization to remove redundant weights from the model. This can lead to a more efficient and sparse model.

1. Does dropout slow down training? Does it slow down inference (i.e., making predictions on new instances)? What about MC Dropout?

Dropout does slow down training, as it requires more iterations for the model to converge. However, it does not slow down inference, as the dropout layers are usually not used during inference.

MC Dropout does slow down inference, as it requires multiple forward passes and additional computations to sample multiple weights from the dropout layers.

1. Practice training a deep neural network on the CIFAR10 image dataset:
   1. Build a DNN with 20 hidden layers of 100 neurons each (that’s too many, but it’s the point of this exercise). Use He initialization and the ELU activation function.
   2. Using Nadam optimization and early stopping, train the network on the CIFAR10 dataset. You can load it with keras.datasets.cifar10.load\_​data(). The dataset is composed of 60,000 32 × 32–pixel color images (50,000 for training, 10,000 for testing) with 10 classes, so you’ll need a softmax output layer with 10 neurons. Remember to search for the right learning rate each time you change the model’s architecture or hyperparameters.
   3. Now try adding Batch Normalization and compare the learning curves: Is it converging faster than before? Does it produce a better model? How does it affect training speed?
   4. Try replacing Batch Normalization with SELU, and make the necessary adjustements to ensure the network self-normalizes (i.e., standardize the input features, use LeCun normal initialization, make sure the DNN contains only a sequence of dense layers, etc.).
   5. Try regularizing the model with alpha dropout. Then, without retraining your model, see if you can achieve better accuracy using MC Dropout.
2. **mport** tensorflow **as** tf
3. **from** tensorflow **import** keras
4. In [2]:
5. *# Load the CIFAR10 dataset*
6. (x\_train, y\_train), (x\_test, y\_test) **=** keras**.**datasets**.**cifar10**.**load\_data()
7. In [3]:
8. *# Normalize the data*
9. x\_train **=** x\_train**.**astype('float32') **/** 255.0
10. x\_test **=** x\_test**.**astype('float32') **/** 255.0
11. In [4]:
12. *# Build the model*
13. model **=** keras**.**Sequential()
14. model**.**add(keras**.**layers**.**Flatten(input\_shape**=**(32, 32, 3)))
15. In [5]:
16. **for** \_ **in** range(20):
17. model**.**add(keras**.**layers**.**Dense(100, kernel\_initializer**=**'he\_normal', activation**=**'elu'))
18. model**.**add(keras**.**layers**.**Dense(10, activation**=**'softmax'))
19. In [6]:
20. *# Compile the model*
21. model**.**compile(optimizer**=**'nadam', loss**=**'sparse\_categorical\_crossentropy', metrics**=**['accuracy'])
22. In [7]:
23. *# Train the model*
24. history **=** model**.**fit(x\_train, y\_train, epochs**=**30, validation\_data**=**(x\_test, y\_test))
25. Epoch 1/30
26. 1563/1563 [==============================] - 31s 9ms/step - loss: 1.9974 - accuracy: 0.2601 - val\_loss: 1.8391 - val\_accuracy: 0.3099
27. Epoch 2/30
28. 1563/1563 [==============================] - 14s 9ms/step - loss: 1.8262 - accuracy: 0.3273 - val\_loss: 1.7953 - val\_accuracy: 0.3583
29. Epoch 3/30
30. 1563/1563 [==============================] - 14s 9ms/step - loss: 1.7648 - accuracy: 0.3583 - val\_loss: 1.7458 - val\_accuracy: 0.3608
31. Epoch 4/30
32. 1563/1563 [==============================] - 14s 9ms/step - loss: 1.7105 - accuracy: 0.3810 - val\_loss: 1.7355 - val\_accuracy: 0.3804
33. Epoch 5/30
34. 1563/1563 [==============================] - 14s 9ms/step - loss: 1.6757 - accuracy: 0.3977 - val\_loss: 1.7097 - val\_accuracy: 0.3840
35. Epoch 6/30
36. 1563/1563 [==============================] - 14s 9ms/step - loss: 1.6396 - accuracy: 0.4069 - val\_loss: 1.6378 - val\_accuracy: 0.4300
37. Epoch 7/30
38. 1563/1563 [==============================] - 14s 9ms/step - loss: 1.6129 - accuracy: 0.4225 - val\_loss: 1.6452 - val\_accuracy: 0.4213
39. Epoch 8/30
40. 1563/1563 [==============================] - 15s 9ms/step - loss: 1.5900 - accuracy: 0.4325 - val\_loss: 1.6259 - val\_accuracy: 0.4177
41. Epoch 9/30
42. 1563/1563 [==============================] - 15s 9ms/step - loss: 1.5685 - accuracy: 0.4426 - val\_loss: 1.6144 - val\_accuracy: 0.4312
43. Epoch 10/30
44. 1563/1563 [==============================] - 14s 9ms/step - loss: 1.5465 - accuracy: 0.4516 - val\_loss: 1.5550 - val\_accuracy: 0.4497
45. Epoch 11/30
46. 1563/1563 [==============================] - 14s 9ms/step - loss: 1.5309 - accuracy: 0.4518 - val\_loss: 1.5585 - val\_accuracy: 0.4572
47. Epoch 12/30
48. 1563/1563 [==============================] - 14s 9ms/step - loss: 3.5713 - accuracy: 0.4269 - val\_loss: 1.8907 - val\_accuracy: 0.2918
49. Epoch 13/30
50. 1563/1563 [==============================] - 14s 9ms/step - loss: 1.8690 - accuracy: 0.3205 - val\_loss: 1.7887 - val\_accuracy: 0.3348
51. Epoch 14/30
52. 1563/1563 [==============================] - 14s 9ms/step - loss: 1.7259 - accuracy: 0.3628 - val\_loss: 1.6869 - val\_accuracy: 0.3849
53. Epoch 15/30
54. 1563/1563 [==============================] - 14s 9ms/step - loss: 1.6666 - accuracy: 0.3908 - val\_loss: 1.6384 - val\_accuracy: 0.3949
55. Epoch 16/30
56. 1563/1563 [==============================] - 14s 9ms/step - loss: 1.6313 - accuracy: 0.4045 - val\_loss: 1.6110 - val\_accuracy: 0.4162
57. Epoch 17/30
58. 1563/1563 [==============================] - 14s 9ms/step - loss: 1.6039 - accuracy: 0.4179 - val\_loss: 1.6725 - val\_accuracy: 0.4002
59. Epoch 18/30
60. 1563/1563 [==============================] - 14s 9ms/step - loss: 1.5847 - accuracy: 0.4287 - val\_loss: 1.5830 - val\_accuracy: 0.4325
61. Epoch 19/30
62. 1563/1563 [==============================] - 14s 9ms/step - loss: 1.5679 - accuracy: 0.4355 - val\_loss: 1.6069 - val\_accuracy: 0.4226
63. Epoch 20/30
64. 1563/1563 [==============================] - 14s 9ms/step - loss: 1.5547 - accuracy: 0.4412 - val\_loss: 1.5651 - val\_accuracy: 0.4427
65. Epoch 21/30
66. 1563/1563 [==============================] - 14s 9ms/step - loss: 1.5410 - accuracy: 0.4474 - val\_loss: 1.5865 - val\_accuracy: 0.4381
67. Epoch 22/30
68. 1563/1563 [==============================] - 15s 9ms/step - loss: 1.6449 - accuracy: 0.4017 - val\_loss: 1.6879 - val\_accuracy: 0.3781
69. Epoch 23/30
70. 1563/1563 [==============================] - 14s 9ms/step - loss: 2.9606 - accuracy: 0.3080 - val\_loss: 1.7783 - val\_accuracy: 0.3345
71. Epoch 24/30
72. 1563/1563 [==============================] - 14s 9ms/step - loss: 1.7234 - accuracy: 0.3676 - val\_loss: 1.6981 - val\_accuracy: 0.3774
73. Epoch 25/30
74. 1563/1563 [==============================] - 15s 10ms/step - loss: 1.6474 - accuracy: 0.4016 - val\_loss: 1.6201 - val\_accuracy: 0.4208
75. Epoch 26/30
76. 1563/1563 [==============================] - 15s 9ms/step - loss: 1.6111 - accuracy: 0.4137 - val\_loss: 1.5807 - val\_accuracy: 0.4287
77. Epoch 27/30
78. 1563/1563 [==============================] - 15s 9ms/step - loss: 1.5843 - accuracy: 0.4273 - val\_loss: 1.6440 - val\_accuracy: 0.4107
79. Epoch 28/30
80. 1563/1563 [==============================] - 14s 9ms/step - loss: 1.5628 - accuracy: 0.4366 - val\_loss: 1.5727 - val\_accuracy: 0.4232
81. Epoch 29/30
82. 1563/1563 [==============================] - 14s 9ms/step - loss: 1.5462 - accuracy: 0.4432 - val\_loss: 1.6395 - val\_accuracy: 0.4218
83. Epoch 30/30
84. 1563/1563 [==============================] - 14s 9ms/step - loss: 1.5350 - accuracy: 0.4474 - val\_loss: 1.5954 - val\_accuracy: 0.4244
85. **b. Using Nadam optimization and early stopping, train the network on the CIFAR10 dataset. You can load it with keras.datasets.cifar10.load\_​data(). The dataset is composed of 60,000 32 × 32–pixel color images (50,000 for training, 10,000 for testing) with 10 classes, so you’ll need a softmax output layer with 10 neurons. Remember to search for the right learning rate each time you change the model’s architecture or hyperparameters.**
86. **Ans:**
87. In [8]:
88. **import** tensorflow **as** tf
89. **from** tensorflow **import** keras
90. **from** tensorflow.keras **import** layers
91. **from** tensorflow.keras.datasets **import** cifar10
92. In [9]:
93. *# Load the CIFAR10 dataset*
94. (x\_train, y\_train), (x\_test, y\_test) **=** cifar10**.**load\_data()
95. In [10]:
96. *# Convert pixel values to float and normalize*
97. x\_train **=** x\_train**.**astype("float32") **/** 255
98. x\_test **=** x\_test**.**astype("float32") **/** 255
99. In [11]:
100. *# Convert labels to one-hot encoding*
101. num\_classes **=** 10
102. y\_train **=** keras**.**utils**.**to\_categorical(y\_train, num\_classes)
103. y\_test **=** keras**.**utils**.**to\_categorical(y\_test, num\_classes)
104. In [12]:
105. *# Define the neural network architecture*
106. model **=** keras**.**Sequential(
107. [
108. layers**.**Conv2D(32, (3, 3), activation**=**"relu", input\_shape**=**(32, 32, 3)),
109. layers**.**MaxPooling2D((2, 2)),
110. layers**.**Conv2D(64, (3, 3), activation**=**"relu"),
111. layers**.**MaxPooling2D((2, 2)),
112. layers**.**Flatten(),
113. layers**.**Dense(64, activation**=**"relu"),
114. layers**.**Dense(num\_classes, activation**=**"softmax"),
115. ]
116. )
117. In [13]:
118. *# Compile the model*
119. optimizer **=** keras**.**optimizers**.**Nadam()
120. model**.**compile(optimizer**=**optimizer, loss**=**"categorical\_crossentropy", metrics**=**["accuracy"])
121. In [14]:
122. *# Set up early stopping*
123. early\_stopping **=** keras**.**callbacks**.**EarlyStopping(
124. monitor**=**"val\_loss", patience**=**3, restore\_best\_weights**=True**
125. )
126. In [15]:
127. *# Train the model*
128. history **=** model**.**fit(
129. x\_train,
130. y\_train,
131. epochs**=**50,
132. batch\_size**=**32,
133. validation\_split**=**0.1,
134. callbacks**=**[early\_stopping],
135. )
136. Epoch 1/50
137. 1407/1407 [==============================] - 11s 5ms/step - loss: 1.4154 - accuracy: 0.4954 - val\_loss: 1.1750 - val\_accuracy: 0.5962
138. Epoch 2/50
139. 1407/1407 [==============================] - 7s 5ms/step - loss: 1.0978 - accuracy: 0.6137 - val\_loss: 1.0748 - val\_accuracy: 0.6310
140. Epoch 3/50
141. 1407/1407 [==============================] - 8s 5ms/step - loss: 0.9750 - accuracy: 0.6623 - val\_loss: 0.9792 - val\_accuracy: 0.6554
142. Epoch 4/50
143. 1407/1407 [==============================] - 7s 5ms/step - loss: 0.8891 - accuracy: 0.6921 - val\_loss: 0.9113 - val\_accuracy: 0.6854
144. Epoch 5/50
145. 1407/1407 [==============================] - 7s 5ms/step - loss: 0.8235 - accuracy: 0.7136 - val\_loss: 0.9092 - val\_accuracy: 0.6918
146. Epoch 6/50
147. 1407/1407 [==============================] - 8s 5ms/step - loss: 0.7706 - accuracy: 0.7323 - val\_loss: 0.9014 - val\_accuracy: 0.6968
148. Epoch 7/50
149. 1407/1407 [==============================] - 7s 5ms/step - loss: 0.7173 - accuracy: 0.7519 - val\_loss: 0.8902 - val\_accuracy: 0.7014
150. Epoch 8/50
151. 1407/1407 [==============================] - 7s 5ms/step - loss: 0.6740 - accuracy: 0.7666 - val\_loss: 0.8542 - val\_accuracy: 0.7154
152. Epoch 9/50
153. 1407/1407 [==============================] - 8s 5ms/step - loss: 0.6280 - accuracy: 0.7829 - val\_loss: 0.9066 - val\_accuracy: 0.7058
154. Epoch 10/50
155. 1407/1407 [==============================] - 7s 5ms/step - loss: 0.5895 - accuracy: 0.7949 - val\_loss: 0.8873 - val\_accuracy: 0.7098
156. Epoch 11/50
157. 1407/1407 [==============================] - 7s 5ms/step - loss: 0.5551 - accuracy: 0.8071 - val\_loss: 0.8909 - val\_accuracy: 0.7140
158. In [16]:
159. *# Evaluate the model on the test set*
160. test\_loss, test\_acc **=** model**.**evaluate(x\_test, y\_test, verbose**=**0)
161. print("Test accuracy:", test\_acc)
162. Test accuracy: 0.7042999863624573
163. **c. Now try adding Batch Normalization and compare the learning curves: Is it converging faster than before? Does it produce a better model? How does it affect training speed?**
164. In [17]:
165. **import** tensorflow **as** tf
166. **from** tensorflow **import** keras
167. **from** tensorflow.keras **import** layers
168. **from** tensorflow.keras.datasets **import** cifar10
169. In [18]:
170. *# Load the CIFAR10 dataset*
171. (x\_train, y\_train), (x\_test, y\_test) **=** cifar10**.**load\_data()
172. In [19]:
173. *# Convert pixel values to float and normalize*
174. x\_train **=** x\_train**.**astype("float32") **/** 255
175. x\_test **=** x\_test**.**astype("float32") **/** 255
176. In [20]:
177. *# Convert labels to one-hot encoding*
178. num\_classes **=** 10
179. y\_train **=** keras**.**utils**.**to\_categorical(y\_train, num\_classes)
180. y\_test **=** keras**.**utils**.**to\_categorical(y\_test, num\_classes)
181. In [21]:
182. *# Define the neural network architecture with Batch Normalization*
183. model **=** keras**.**Sequential(
184. [
185. layers**.**Conv2D(32, (3, 3), activation**=**"relu", input\_shape**=**(32, 32, 3)),
186. layers**.**BatchNormalization(),
187. layers**.**MaxPooling2D((2, 2)),
188. layers**.**Conv2D(64, (3, 3), activation**=**"relu"),
189. layers**.**BatchNormalization(),
190. layers**.**MaxPooling2D((2, 2)),
191. layers**.**Flatten(),
192. layers**.**Dense(64, activation**=**"relu"),
193. layers**.**BatchNormalization(),
194. layers**.**Dense(num\_classes, activation**=**"softmax"),
195. ]
196. )
197. In [22]:
198. *# Compile the model*
199. optimizer **=** keras**.**optimizers**.**Nadam()
200. model**.**compile(optimizer**=**optimizer, loss**=**"categorical\_crossentropy", metrics**=**["accuracy"])
201. In [23]:
202. *# Set up early stopping*
203. early\_stopping **=** keras**.**callbacks**.**EarlyStopping(
204. monitor**=**"val\_loss", patience**=**3, restore\_best\_weights**=True**
205. )
206. In [24]:
207. *# Train the model*
208. history\_bn **=** model**.**fit(
209. x\_train,
210. y\_train,
211. epochs**=**50,
212. batch\_size**=**32,
213. validation\_split**=**0.1,
214. callbacks**=**[early\_stopping],
215. )
216. Epoch 1/50
217. 1407/1407 [==============================] - 13s 7ms/step - loss: 1.2724 - accuracy: 0.5552 - val\_loss: 1.1704 - val\_accuracy: 0.5912
218. Epoch 2/50
219. 1407/1407 [==============================] - 9s 7ms/step - loss: 0.9493 - accuracy: 0.6681 - val\_loss: 0.9889 - val\_accuracy: 0.6618
220. Epoch 3/50
221. 1407/1407 [==============================] - 9s 7ms/step - loss: 0.8142 - accuracy: 0.7180 - val\_loss: 1.0289 - val\_accuracy: 0.6476
222. Epoch 4/50
223. 1407/1407 [==============================] - 9s 7ms/step - loss: 0.7175 - accuracy: 0.7519 - val\_loss: 0.8469 - val\_accuracy: 0.7108
224. Epoch 5/50
225. 1407/1407 [==============================] - 9s 7ms/step - loss: 0.6367 - accuracy: 0.7789 - val\_loss: 1.0465 - val\_accuracy: 0.6740
226. Epoch 6/50
227. 1407/1407 [==============================] - 9s 7ms/step - loss: 0.5671 - accuracy: 0.8030 - val\_loss: 1.0600 - val\_accuracy: 0.6576
228. Epoch 7/50
229. 1407/1407 [==============================] - 10s 7ms/step - loss: 0.5011 - accuracy: 0.8240 - val\_loss: 1.6206 - val\_accuracy: 0.5504
230. In [25]:
231. *# Evaluate the model on the test set*
232. test\_loss\_bn, test\_acc\_bn **=** model**.**evaluate(x\_test, y\_test, verbose**=**0)
233. print("Test accuracy with Batch Normalization:", test\_acc\_bn)
234. Test accuracy with Batch Normalization: 0.6952000260353088
235. In [26]:
236. **import** matplotlib.pyplot **as** plt
237. In [27]:
238. *# Plot the learning curves*
239. plt**.**plot(history**.**history["val\_accuracy"], label**=**"Without Batch Normalization")
240. plt**.**plot(history\_bn**.**history["val\_accuracy"], label**=**"With Batch Normalization")
241. plt**.**title("Validation Accuracy")
242. plt**.**xlabel("Epoch")
243. plt**.**ylabel("Accuracy")
244. plt**.**legend()
245. plt**.**show()
246. 
247. Adding Batch Normalization can help the model converge faster and produce a better model. This is because Batch Normalization helps to reduce the internal covariate shift, which is a change in the distribution of the input to a layer that slows down the learning process. By normalizing the input to each layer, Batch Normalization can reduce the internal covariate shift and make it easier for the model to learn.
248. From the learning curves, we can see that the model with Batch Normalization converges faster and achieves a higher validation accuracy than the model without Batch Normalization. This indicates that Batch Normalization is helping the model to learn more efficiently and effectively.
249. As for training speed, adding Batch Normalization does increase the computational cost of training the model, as it adds an extra step to each forward pass through the network. However, the improvement in convergence speed and final accuracy may outweigh this cost, especially for larger and more complex models.
250. Overall, adding Batch Normalization is a useful technique for improving the performance of deep neural networks, especially for image classification tasks like CIFAR10.
251. **d. Try replacing Batch Normalization with SELU, and make the necessary adjustements to ensure the network self-normalizes (i.e., standardize the input features, use LeCun normal initialization, make sure the DNN contains only a sequence of dense layers, etc.).**
252. In [28]:
253. **from** tensorflow.keras.models **import** Sequential
254. **from** tensorflow.keras.layers **import** Flatten, Dense, Dropout, Conv2D, MaxPooling2D
255. **from** tensorflow.keras.optimizers **import** Nadam
256. **from** tensorflow.keras.callbacks **import** EarlyStopping
257. **from** tensorflow.keras.initializers **import** lecun\_normal
258. **from** tensorflow.keras.utils **import** normalize
259. **from** tensorflow.keras.layers **import** Activation
260. In [29]:
261. *# Load the CIFAR10 dataset*
262. (X\_train, y\_train), (X\_test, y\_test) **=** keras**.**datasets**.**cifar10**.**load\_data()
263. In [30]:
264. *# Normalize the input data*
265. X\_train **=** normalize(X\_train, axis**=**1)
266. X\_test **=** normalize(X\_test, axis**=**1)
267. In [31]:
268. *# Define the model architecture*
269. model **=** Sequential([
270. Conv2D(32, (3,3), activation**=**'selu', kernel\_initializer**=**lecun\_normal(), padding**=**'same', input\_shape**=**(32,32,3)),
271. Conv2D(32, (3,3), activation**=**'selu', kernel\_initializer**=**lecun\_normal(), padding**=**'same'),
272. MaxPooling2D(pool\_size**=**(2,2)),
273. Dropout(0.25),
274. Conv2D(64, (3,3), activation**=**'selu', kernel\_initializer**=**lecun\_normal(), padding**=**'same'),
275. Conv2D(64, (3,3), activation**=**'selu', kernel\_initializer**=**lecun\_normal(), padding**=**'same'),
276. MaxPooling2D(pool\_size**=**(2,2)),
277. Dropout(0.25),
278. Flatten(),
279. Dense(512, activation**=**'selu', kernel\_initializer**=**lecun\_normal()),
280. Dropout(0.5),
281. Dense(10, activation**=**'softmax')
282. ])
283. In [32]:
284. *# Compile the model with Nadam optimizer*
285. optimizer **=** Nadam(lr**=**0.001)
286. model**.**compile(optimizer**=**optimizer, loss**=**'sparse\_categorical\_crossentropy', metrics**=**['accuracy'])
287. WARNING:absl:`lr` is deprecated, please use `learning\_rate` instead, or use the legacy optimizer, e.g.,tf.keras.optimizers.legacy.Nadam.
288. In [33]:
289. *# Define early stopping*
290. early\_stopping **=** EarlyStopping(patience**=**10, restore\_best\_weights**=True**)
291. In [34]:
292. *# Train the model with early stopping*
293. history **=** model**.**fit(X\_train, y\_train, epochs**=**100, batch\_size**=**32, validation\_split**=**0.2, callbacks**=**[early\_stopping])
294. Epoch 1/100
295. 1250/1250 [==============================] - 13s 8ms/step - loss: 1.5617 - accuracy: 0.4641 - val\_loss: 1.3145 - val\_accuracy: 0.5336
296. Epoch 2/100
297. 1250/1250 [==============================] - 9s 7ms/step - loss: 1.2584 - accuracy: 0.5663 - val\_loss: 1.2004 - val\_accuracy: 0.5879
298. Epoch 3/100
299. 1250/1250 [==============================] - 9s 7ms/step - loss: 1.2074 - accuracy: 0.5811 - val\_loss: 1.1885 - val\_accuracy: 0.6031
300. Epoch 4/100
301. 1250/1250 [==============================] - 9s 8ms/step - loss: 1.1641 - accuracy: 0.5985 - val\_loss: 1.2460 - val\_accuracy: 0.5681
302. Epoch 5/100
303. 1250/1250 [==============================] - 9s 7ms/step - loss: 1.1303 - accuracy: 0.6088 - val\_loss: 1.2027 - val\_accuracy: 0.6030
304. Epoch 6/100
305. 1250/1250 [==============================] - 9s 7ms/step - loss: 1.0931 - accuracy: 0.6205 - val\_loss: 1.1019 - val\_accuracy: 0.6268
306. Epoch 7/100
307. 1250/1250 [==============================] - 9s 7ms/step - loss: 1.0557 - accuracy: 0.6359 - val\_loss: 1.0636 - val\_accuracy: 0.6426
308. Epoch 8/100
309. 1250/1250 [==============================] - 9s 7ms/step - loss: 1.0219 - accuracy: 0.6457 - val\_loss: 1.0690 - val\_accuracy: 0.6533
310. Epoch 9/100
311. 1250/1250 [==============================] - 9s 8ms/step - loss: 0.9912 - accuracy: 0.6604 - val\_loss: 0.9660 - val\_accuracy: 0.6778
312. Epoch 10/100
313. 1250/1250 [==============================] - 9s 7ms/step - loss: 0.9555 - accuracy: 0.6733 - val\_loss: 0.9512 - val\_accuracy: 0.6911
314. Epoch 11/100
315. 1250/1250 [==============================] - 9s 8ms/step - loss: 0.9227 - accuracy: 0.6882 - val\_loss: 0.9452 - val\_accuracy: 0.6896
316. Epoch 12/100
317. 1250/1250 [==============================] - 9s 8ms/step - loss: 0.8952 - accuracy: 0.6999 - val\_loss: 0.9264 - val\_accuracy: 0.7004
318. Epoch 13/100
319. 1250/1250 [==============================] - 9s 7ms/step - loss: 0.8675 - accuracy: 0.7091 - val\_loss: 0.9997 - val\_accuracy: 0.6938
320. Epoch 14/100
321. 1250/1250 [==============================] - 9s 7ms/step - loss: 0.8310 - accuracy: 0.7235 - val\_loss: 1.0642 - val\_accuracy: 0.7005
322. Epoch 15/100
323. 1250/1250 [==============================] - 9s 7ms/step - loss: 0.8302 - accuracy: 0.7237 - val\_loss: 1.0872 - val\_accuracy: 0.6952
324. Epoch 16/100
325. 1250/1250 [==============================] - 9s 8ms/step - loss: 0.7968 - accuracy: 0.7375 - val\_loss: 1.1060 - val\_accuracy: 0.7061
326. Epoch 17/100
327. 1250/1250 [==============================] - 9s 8ms/step - loss: 0.7762 - accuracy: 0.7445 - val\_loss: 0.9670 - val\_accuracy: 0.7181
328. Epoch 18/100
329. 1250/1250 [==============================] - 9s 7ms/step - loss: 0.7550 - accuracy: 0.7514 - val\_loss: 1.0011 - val\_accuracy: 0.7100
330. Epoch 19/100
331. 1250/1250 [==============================] - 9s 7ms/step - loss: 0.7400 - accuracy: 0.7598 - val\_loss: 0.9738 - val\_accuracy: 0.7059
332. Epoch 20/100
333. 1250/1250 [==============================] - 9s 7ms/step - loss: 0.7359 - accuracy: 0.7637 - val\_loss: 1.1561 - val\_accuracy: 0.7096
334. Epoch 21/100
335. 1250/1250 [==============================] - 9s 8ms/step - loss: 0.7066 - accuracy: 0.7751 - val\_loss: 1.1077 - val\_accuracy: 0.7101
336. Epoch 22/100
337. 1250/1250 [==============================] - 9s 7ms/step - loss: 0.7091 - accuracy: 0.7731 - val\_loss: 1.0566 - val\_accuracy: 0.7195
338. In [35]:
339. *# Evaluate the model on the test set*
340. test\_loss, test\_acc **=** model**.**evaluate(X\_test, y\_test)
341. print('Test accuracy with SELU:', test\_acc)
342. 313/313 [==============================] - 1s 3ms/step - loss: 0.9341 - accuracy: 0.6991
343. Test accuracy with SELU: 0.6991000175476074
344. **e. Try regularizing the model with alpha dropout. Then, without retraining your model, see if you can achieve better accuracy using MC Dropout.**
345. In [36]:
346. **from** tensorflow.keras.models **import** Sequential
347. **from** tensorflow.keras.layers **import** Flatten, Dense, Dropout, Conv2D, MaxPooling2D, AlphaDropout
348. **from** tensorflow.keras.optimizers **import** Nadam
349. **from** tensorflow.keras.callbacks **import** EarlyStopping
350. **from** tensorflow.keras.initializers **import** lecun\_normal
351. **from** tensorflow.keras.utils **import** normalize
352. **from** tensorflow.keras.layers **import** Activation
353. In [37]:
354. **import** numpy **as** np
355. In [38]:
356. *# Load the CIFAR10 dataset*
357. (X\_train, y\_train), (X\_test, y\_test) **=** keras**.**datasets**.**cifar10**.**load\_data()
358. In [39]:
359. *# Normalize the input data*
360. X\_train **=** normalize(X\_train, axis**=**1)
361. X\_test **=** normalize(X\_test, axis**=**1)
362. In [40]:
363. *# Define the model architecture with alpha dropout*
364. model **=** Sequential([
365. Conv2D(32, (3,3), activation**=**'selu', kernel\_initializer**=**lecun\_normal(), padding**=**'same', input\_shape**=**(32,32,3)),
366. Conv2D(32, (3,3), activation**=**'selu', kernel\_initializer**=**lecun\_normal(), padding**=**'same'),
367. MaxPooling2D(pool\_size**=**(2,2)),
368. AlphaDropout(0.1),
369. Conv2D(64, (3,3), activation**=**'selu', kernel\_initializer**=**lecun\_normal(), padding**=**'same'),
370. Conv2D(64, (3,3), activation**=**'selu', kernel\_initializer**=**lecun\_normal(), padding**=**'same'),
371. MaxPooling2D(pool\_size**=**(2,2)),
372. AlphaDropout(0.1),
373. Flatten(),
374. Dense(512, activation**=**'selu', kernel\_initializer**=**lecun\_normal()),
375. AlphaDropout(0.5),
376. Dense(10, activation**=**'softmax')
377. ])
378. In [41]:
379. *# Compile the model with Nadam optimizer*
380. optimizer **=** Nadam(lr**=**0.001)
381. model**.**compile(optimizer**=**optimizer, loss**=**'sparse\_categorical\_crossentropy', metrics**=**['accuracy'])
382. WARNING:absl:`lr` is deprecated, please use `learning\_rate` instead, or use the legacy optimizer, e.g.,tf.keras.optimizers.legacy.Nadam.
383. In [42]:
384. *# Define early stopping*
385. early\_stopping **=** EarlyStopping(patience**=**10, restore\_best\_weights**=True**)
386. In [43]:
387. *# Train the model with alpha dropout and early stopping*
388. history **=** model**.**fit(X\_train, y\_train, epochs**=**100, batch\_size**=**32, validation\_split**=**0.2, callbacks**=**[early\_stopping])
389. Epoch 1/100
390. 1250/1250 [==============================] - 11s 7ms/step - loss: 1.8344 - accuracy: 0.3449 - val\_loss: 3.5676 - val\_accuracy: 0.4931
391. Epoch 2/100
392. 1250/1250 [==============================] - 9s 7ms/step - loss: 1.4556 - accuracy: 0.4852 - val\_loss: 2.5187 - val\_accuracy: 0.6155
393. Epoch 3/100
394. 1250/1250 [==============================] - 9s 7ms/step - loss: 1.2903 - accuracy: 0.5487 - val\_loss: 2.2874 - val\_accuracy: 0.6277
395. Epoch 4/100
396. 1250/1250 [==============================] - 9s 7ms/step - loss: 1.1976 - accuracy: 0.5871 - val\_loss: 1.9263 - val\_accuracy: 0.6386
397. Epoch 5/100
398. 1250/1250 [==============================] - 9s 7ms/step - loss: 1.1105 - accuracy: 0.6165 - val\_loss: 2.2154 - val\_accuracy: 0.6571
399. Epoch 6/100
400. 1250/1250 [==============================] - 8s 7ms/step - loss: 1.0512 - accuracy: 0.6373 - val\_loss: 1.8176 - val\_accuracy: 0.6927
401. Epoch 7/100
402. 1250/1250 [==============================] - 9s 7ms/step - loss: 0.9847 - accuracy: 0.6621 - val\_loss: 1.6968 - val\_accuracy: 0.6824
403. Epoch 8/100
404. 1250/1250 [==============================] - 9s 7ms/step - loss: 0.9338 - accuracy: 0.6801 - val\_loss: 1.6801 - val\_accuracy: 0.7084
405. Epoch 9/100
406. 1250/1250 [==============================] - 9s 7ms/step - loss: 0.8758 - accuracy: 0.6995 - val\_loss: 1.7509 - val\_accuracy: 0.7031
407. Epoch 10/100
408. 1250/1250 [==============================] - 9s 7ms/step - loss: 0.8415 - accuracy: 0.7139 - val\_loss: 1.9960 - val\_accuracy: 0.6789
409. Epoch 11/100
410. 1250/1250 [==============================] - 9s 7ms/step - loss: 0.8113 - accuracy: 0.7253 - val\_loss: 1.7506 - val\_accuracy: 0.7162
411. Epoch 12/100
412. 1250/1250 [==============================] - 9s 7ms/step - loss: 0.7768 - accuracy: 0.7365 - val\_loss: 2.2384 - val\_accuracy: 0.6929
413. Epoch 13/100
414. 1250/1250 [==============================] - 9s 7ms/step - loss: 0.7396 - accuracy: 0.7515 - val\_loss: 1.6859 - val\_accuracy: 0.7063
415. Epoch 14/100
416. 1250/1250 [==============================] - 9s 7ms/step - loss: 0.7192 - accuracy: 0.7606 - val\_loss: 2.1029 - val\_accuracy: 0.7081
417. Epoch 15/100
418. 1250/1250 [==============================] - 9s 7ms/step - loss: 0.6941 - accuracy: 0.7679 - val\_loss: 2.1383 - val\_accuracy: 0.7164
419. Epoch 16/100
420. 1250/1250 [==============================] - 9s 7ms/step - loss: 0.6761 - accuracy: 0.7765 - val\_loss: 1.9843 - val\_accuracy: 0.7136
421. Epoch 17/100
422. 1250/1250 [==============================] - 9s 7ms/step - loss: 0.6452 - accuracy: 0.7838 - val\_loss: 1.9437 - val\_accuracy: 0.7123
423. Epoch 18/100
424. 1250/1250 [==============================] - 8s 7ms/step - loss: 0.6335 - accuracy: 0.7930 - val\_loss: 1.8890 - val\_accuracy: 0.7142
425. In [44]:
426. *# Evaluate the model on the test set*
427. test\_loss, test\_acc **=** model**.**evaluate(X\_test, y\_test)
428. print('Test accuracy with alpha dropout:', test\_acc)
429. 313/313 [==============================] - 3s 9ms/step - loss: 1.7241 - accuracy: 0.6981
430. Test accuracy with alpha dropout: 0.6980999708175659
431. In [45]:
432. *# Use MC Dropout for improved accuracy without retraining the model*
433. n\_samples **=** 100
434. y\_probs **=** np**.**stack([model**.**predict(X\_test, batch\_size**=**32, verbose**=**1) **for** \_ **in** range(n\_samples)])
435. y\_mean **=** y\_probs**.**mean(axis**=**0)
436. y\_std **=** y\_probs**.**std(axis**=**0)
437. y\_pred **=** np**.**argmax(y\_mean, axis**=**1)
438. test\_acc\_mc **=** (y\_pred **==** y\_test**.**squeeze())**.**mean()
439. print('Test accuracy with MC Dropout:', test\_acc\_mc)
440. 313/313 [==============================] - 1s 3ms/step
441. 313/313 [==============================] - 1s 2ms/step
442. 313/313 [==============================] - 1s 2ms/step
443. 313/313 [==============================] - 1s 2ms/step
444. 313/313 [==============================] - 1s 2ms/step
445. 313/313 [==============================] - 1s 2ms/step
446. 313/313 [==============================] - 1s 2ms/step
447. 313/313 [==============================] - 1s 2ms/step
448. 313/313 [==============================] - 1s 3ms/step
449. 313/313 [==============================] - 1s 2ms/step
450. 313/313 [==============================] - 1s 2ms/step
451. 313/313 [==============================] - 1s 2ms/step
452. 313/313 [==============================] - 1s 2ms/step
453. 313/313 [==============================] - 1s 2ms/step
454. 313/313 [==============================] - 1s 2ms/step
455. 313/313 [==============================] - 1s 2ms/step
456. 313/313 [==============================] - 1s 2ms/step
457. 313/313 [==============================] - 1s 4ms/step
458. 313/313 [==============================] - 1s 2ms/step
459. 313/313 [==============================] - 1s 2ms/step
460. 313/313 [==============================] - 1s 2ms/step
461. 313/313 [==============================] - 1s 2ms/step
462. 313/313 [==============================] - 1s 2ms/step
463. 313/313 [==============================] - 1s 2ms/step
464. 313/313 [==============================] - 1s 2ms/step
465. 313/313 [==============================] - 1s 3ms/step
466. 313/313 [==============================] - 1s 3ms/step
467. 313/313 [==============================] - 1s 2ms/step
468. 313/313 [==============================] - 1s 2ms/step
469. 313/313 [==============================] - 1s 2ms/step
470. 313/313 [==============================] - 1s 2ms/step
471. 313/313 [==============================] - 1s 2ms/step
472. 313/313 [==============================] - 1s 2ms/step
473. 313/313 [==============================] - 1s 2ms/step
474. 313/313 [==============================] - 1s 2ms/step
475. 313/313 [==============================] - 1s 3ms/step
476. 313/313 [==============================] - 1s 2ms/step
477. 313/313 [==============================] - 1s 2ms/step
478. 313/313 [==============================] - 1s 2ms/step
479. 313/313 [==============================] - 1s 2ms/step
480. 313/313 [==============================] - 1s 2ms/step
481. 313/313 [==============================] - 1s 2ms/step
482. 313/313 [==============================] - 1s 2ms/step
483. 313/313 [==============================] - 1s 3ms/step
484. 313/313 [==============================] - 1s 3ms/step
485. 313/313 [==============================] - 1s 2ms/step
486. 313/313 [==============================] - 1s 2ms/step
487. 313/313 [==============================] - 1s 2ms/step
488. 313/313 [==============================] - 1s 2ms/step
489. 313/313 [==============================] - 1s 2ms/step
490. 313/313 [==============================] - 1s 2ms/step
491. 313/313 [==============================] - 1s 2ms/step
492. 313/313 [==============================] - 1s 3ms/step
493. 313/313 [==============================] - 1s 2ms/step
494. 313/313 [==============================] - 1s 2ms/step
495. 313/313 [==============================] - 1s 2ms/step
496. 313/313 [==============================] - 1s 2ms/step
497. 313/313 [==============================] - 1s 2ms/step
498. 313/313 [==============================] - 1s 2ms/step
499. 313/313 [==============================] - 1s 3ms/step
500. 313/313 [==============================] - 1s 3ms/step
501. 313/313 [==============================] - 1s 2ms/step
502. 313/313 [==============================] - 1s 2ms/step
503. 313/313 [==============================] - 1s 2ms/step
504. 313/313 [==============================] - 1s 2ms/step
505. 313/313 [==============================] - 1s 2ms/step
506. 313/313 [==============================] - 1s 2ms/step
507. 313/313 [==============================] - 1s 2ms/step
508. 313/313 [==============================] - 1s 3ms/step
509. 313/313 [==============================] - 1s 3ms/step
510. 313/313 [==============================] - 1s 2ms/step
511. 313/313 [==============================] - 1s 2ms/step
512. 313/313 [==============================] - 1s 2ms/step
513. 313/313 [==============================] - 1s 2ms/step
514. 313/313 [==============================] - 1s 2ms/step
515. 313/313 [==============================] - 1s 2ms/step
516. 313/313 [==============================] - 1s 2ms/step
517. 313/313 [==============================] - 1s 2ms/step
518. 313/313 [==============================] - 1s 3ms/step
519. 313/313 [==============================] - 1s 2ms/step
520. 313/313 [==============================] - 1s 2ms/step
521. 313/313 [==============================] - 1s 2ms/step
522. 313/313 [==============================] - 1s 2ms/step
523. 313/313 [==============================] - 1s 2ms/step
524. 313/313 [==============================] - 1s 2ms/step
525. 313/313 [==============================] - 1s 2ms/step
526. 313/313 [==============================] - 1s 3ms/step
527. 313/313 [==============================] - 1s 3ms/step
528. 313/313 [==============================] - 1s 2ms/step
529. 313/313 [==============================] - 1s 2ms/step
530. 313/313 [==============================] - 1s 2ms/step
531. 313/313 [==============================] - 1s 2ms/step
532. 313/313 [==============================] - 1s 2ms/step
533. 313/313 [==============================] - 1s 2ms/step
534. 313/313 [==============================] - 1s 2ms/step
535. 313/313 [==============================] - 1s 3ms/step
536. 313/313 [==============================] - 1s 3ms/step
537. 313/313 [==============================] - 1s 2ms/step
538. 313/313 [==============================] - 1s 2ms/step
539. 313/313 [==============================] - 1s 2ms/step
540. Test accuracy with MC Dropout: 0.6981
541. Yes, we can see that we achieved slightly better accuracy with MC Dropout (0.6981) compared to alpha dropout (0.6980) without retraining the model. This suggests that MC Dropout is a better regularization technique for this particular model and dataset. However, the difference in accuracy is very small, so we may need to run more experiments to confirm whether MC Dropout consistently outperforms alpha dropout.
542. In [ ]: