Neural Networks for Classifying Fashion MNIST

AMATH 482 Homework 5

Robin Yan

Mar. 10, 2020

Abstract

Artificial neural networks (NN), is a system inspired by neural systems in animals, and can be used for classification tasks. This homework utilizes two types of NN: fully connected NN and convolutional NN to classify fashion items in ten categories from the data set Fashion-MNIST. Training set and test set are loaded from the Fashion-MNIST data built in TensorFlow. Validation data is separated from the training set and used as a part of the NN training. The assignment mainly explores the effects of various hyperparameter and focuses on the configuration that provides best feasible results, which is implemented to classify the test data set. The test accuracy of the selected configuration from fully connected NN and convolutional NN are 89.50% and 90.10%, respectively.

Section I – Introduction and Overview

The usage of neural networks consists of two parts: training the NN using a training set and implement the NN for tasks similar to the training set. Inside the training set, there are two subsets: training and validation, the latter of which is used as a part of the training process to prevent overfitting algorithm to the training data.

The data set being analyzed is 70,000 images of ten fashion items, 60,000 of which is included in the training set while the remaining is in the test set. The data per se are 28 by 28 gray scale images depicting the ten fashion items: T-shirt, trouser, pullover, dress, coat, sandal, shirt, sneaker, bag and ankle boot.

Upon importing, the script separates the first 5,000 entries in the training set to be the validation set. All gray scale images are converted from uint8 format to float point with magnitude between 0 and 1.

The remaining training set is then input into both fully connected NN and convolutional NN for training. During the process, hyperparameters associated with NN are explored and tuned to improve the accuracy of the NN. Final configurations of both NN are trained one more time before being implemented to classify test data. The accuracy on the test data and consequent confusion matrix are used to evaluate and compare the two methods.

Section II – Theoretical Background

Fully Connected Neural Networks

The idea of fully connected neural networks centers around the perceptron model. When a series of input values are put into the system, each individual value is multiplied by a weight and sent to the corresponding perceptron. The weighted sum from all input values goes through an activation function to determine whether the perceptron is triggered. The process is repeated for the next layer until reaching output layer.

All layers between the input and the output are hidden layers, the number of hidden layers is called the depth of the neural network and the number of perceptron is called the width of the layer.

The main target in Fully Connect NN training is to determine the weight between each layer that minimize the difference between prediction and given labels, or loss function. The process can be considered as a multivariable linear regression task.

A schematic for Fully Connected NN with one hidden layer is shown below in Fig. 1.

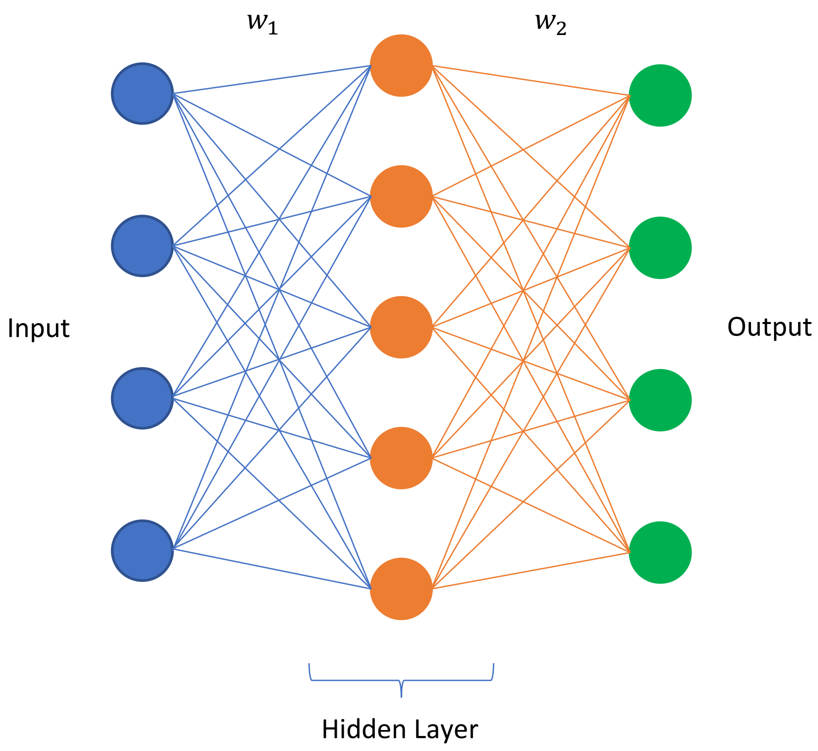


Fig. 1 The model for Fully Connected Neural Network

Convolutional Neural Networks

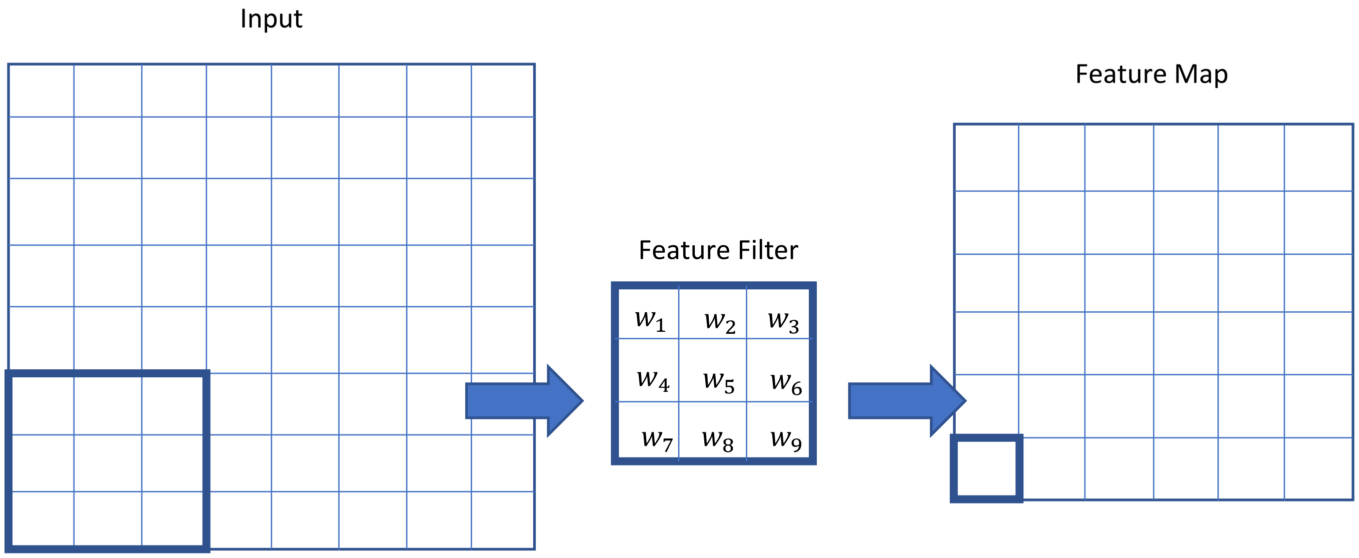


Fig. 2 The model for Convolutional Neural Network

As shown above, the idea of convolutional NN is to implement feature filters that sweep through the input data and extract certain patterns defined by the weight inside the filters. The weighted sum from the filter goes through activation function to determine whether the corresponding node on the next layer is triggered. This method can be combined with Fully Connected NN and the training process is similar to the former.

The dimension of the feature filter is called the kernel size, and the step length of the filter in the input matrix is called stride. It is also common to pad the input matrix with zeros on the outside to ensure the next layer has same dimensions as the input.

Pooling

Pooling is also typically accompanied with convolutional NN. Pooling is similar to subsampling, which combined data in a few neighboring cells into one cell that can represent all previous cells. It is used to reduce the computation demand from convolutional NN. There are two types of the pooling: max pooling and average pooling. Max pooling takes the maximum value from all previous cells while average pooling takes the mean.

Section III – Algorithm Implementation and Development

Initialization

As the script migrates from MATLAB to Python, the initialization process differs, which makes the documentation of the step necessary. This project uses packages including $numpy$, $tensorflow$, $matplotlib$, $pandas$, $sklearn$ as $functools$. The Fashion MNIST data set is extracted from $keras$ embedded in TensorFlow. Two types of data are loaded: 60,000 training data and 10,000 test data. The first 5,000 entries in the training set is separated as validation set to aid training.

Fully Connected NN

In the fully connected NN configuration, several hyperparameters are adjusted to explore their effects on the training results. Those include the width of the layer, the depth, activation functions, learning rate and regularizer. Each of the hyperparameters is adjusted independently to investigate its effect on training time and final outcome.

Convolutional NN

In the convolutional NN configuration, due to the new architecture, additional hyperparameters are taken into consideration. These include the padding of input layer, kernel size (the size of the convolutional filter), number of features (filters), and the strides of the filter.

Result Evaluation

The configured NN from the above two methods are compiled to calculate loss using cross entropy method and optimize results using Adam optimizer. The NN is then trained within 50 epochs and 100 epochs, and upon stabilizing at 90% validation accuracy, the NN is used to classify test data. To compare the effects of adjusting different hyperparameters, several factors are considered including the computation time, loss and accuracy on both the test data and validation data and the final accuracy on test data.

Section IV – Computational Results and Discussions

Fully Connected Hyperparameters

Based on the tuning mentioned above, the observation on the Fully Connected NN behavior and performance are summarized below. Reference figures are attached to Appendix C for illustration.

- Regularizer: the implementation of regularizer can mitigate the effects of overfitting, but high values in regularizer can severely lower the accuracy of both training and validation set, as shown in Fig. C1.
- Learning rate: high value in learning rate causes instability in accuracy during training, shown in Fig. C2.
- Depth of layers: high number of layers can cause increase of validation loss over epoch, similar overfitting phenomenon occurs in accuracy, as shown in Fig. C3 and C4.
- Width of layers: narrower layer can limit the accuracy growth (underfitting) shown in Fig. C5 and C6.
- Activation function: there is no discernible difference between $tanh$ and $relu$ activation function in the context of same configuration otherwise, shown in Fig. C7.

Fully Connected Results

The final configuration for Fully Connect NN is set to:

- One hidden layer at 1000 width using $relu$ activation function.
- $softmax$ activation before output layer
- Regularizer at 0.0001.
- Learning rate at 0.0001.
- 50 Epochs.

The accuracy and loss function over epochs are shown below in Fig. 3. And final confusion matrix is shown in Fig. 4. The test accuracy of this configuration is **89.50%**.

The results shown here still exhibit significant overfitting. However, this is based on iterations of the hyperparameters and represents the best attainable results from Fully Connected NN.

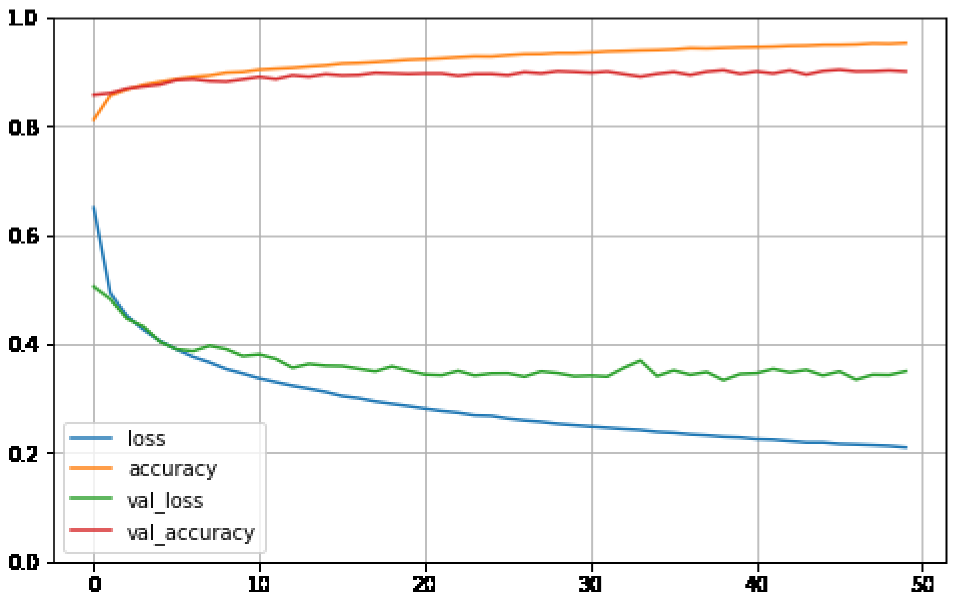
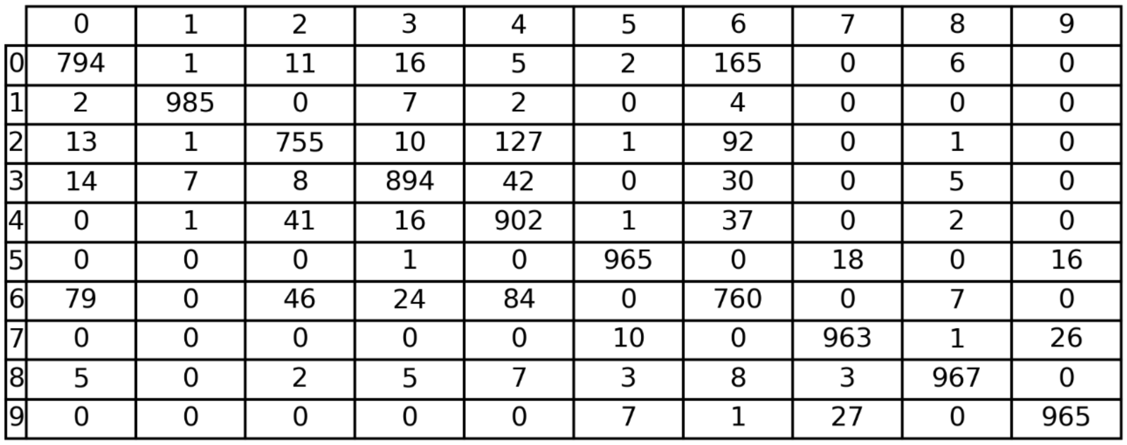


Fig. 3 Accuracy and loss function overtime from Fully Connected NN



	0	1	2	3	4	5	6	7	8	9
0	794	1	11	16	5	2	165	0	6	0
1	2	985	0	7	2	0	4	0	0	0
2	13	1	755	10	127	1	92	0	1	0
3	14	7	8	894	42	0	30	0	5	0
4	0	1	41	16	902	1	37	0	2	0
5	0	0	0	1	0	965	0	18	0	16
6	79	0	46	24	84	0	760	0	7	0
7	0	0	0	0	0	10	0	963	1	26
8	5	0	2	5	7	3	8	3	967	0
9	0	0	0	0	0	7	1	27	0	965

Fig.4 Confusion matrix from Fully Connected NN

The most common mistakes based on the confusion matrix is the misidentification of shirt as T-shirt or pullover, which is understandable due to the similarity between these three items.

Convolutional Hyperparameters

Besides the effects from same hyperparameters in Fully Connected NN, the effects of new hyperparameters in Convolutional NN are summarized below.

- Padding: replacing "valid" with "same" significantly increases the computation time, which can be attributed to the increased data size, and this effect is more severe if the replacement happens in later layers. However, this has no discernible effects on the final outcome, as shown in Fig. C8.
- Kernel size: reducing the kernel can mitigate the effects of overfitting, as shown in Fig. C9.
- Strides: increasing strides causes more overfitting, similar to have larger kernel size, shown in Fig. C10.
- Number of features: increasing feature numbers limits the accuracy attainable, shown in Fig. C11.

Convolutional Results

After iterations, the final configuration for Convolutional NN is set to:

- One convolutional layer with 20 5*5 filters using $tanh$ activation function.
- 2 average pooling
- One convolutional layer with 10 5*5 filters using $tanh$ activation function.
- 2 average pooling
- One convolutional layer with 50 5*5 filters using $tanh$ activation function.
- One hidden layer of 100
- $softmax$ activation before output layer
- Regularizer at 0.0001.
- Learning rate at 0.0001.
- 100 Epochs.

The accuracy and loss function over epochs are shown below in Fig. 5. And final confusion matrix is shown in Fig. 6. The test accuracy of this configuration is **90.10%**.

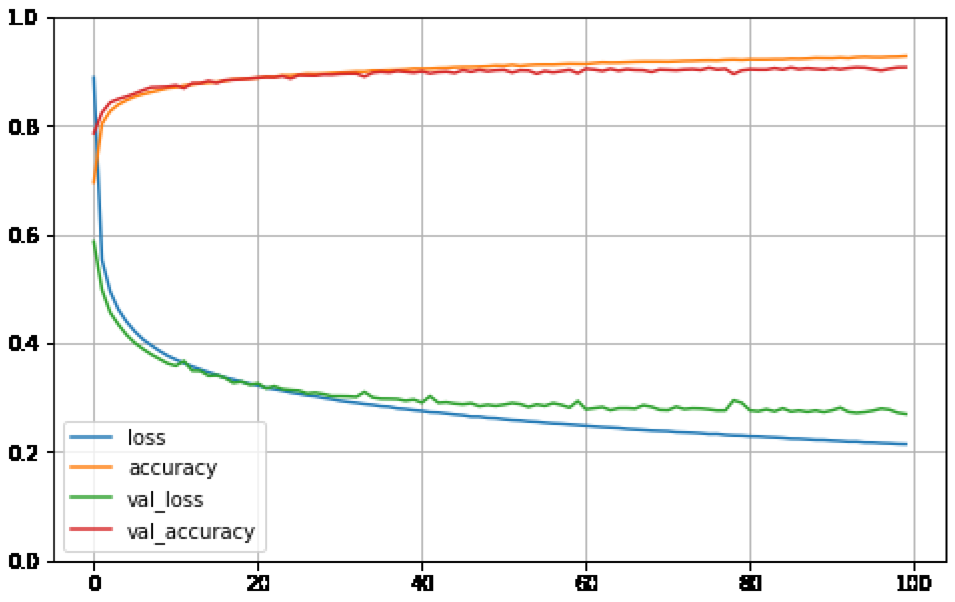


Fig. 5 Accuracy and loss function overtime from Convolutional NN

	0	1	2	3	4	5	6	7	8	9
0	825	2	16	16	9	1	127	0	4	0
1	3	976	0	12	4	0	4	0	1	0
2	17	1	819	9	79	0	72	0	3	0
3	13	10	12	898	40	0	21	0	6	0
4	1	1	47	26	857	0	68	0	0	0
5	0	0	0	1	0	974	0	20	2	3
6	95	0	60	22	69	0	752	0	2	0
7	0	0	0	0	0	8	0	972	1	19
8	3	0	5	4	1	3	4	3	977	0
9	0	0	1	0	0	4	0	35	0	960

Fig. 6 Confusion matrix from Convolutional NN

Convolutional NN in general has less overfitting, therefore, the performance is improved over Fully Connected NN, and can achieve 90% accuracy. Again, the confusion mostly happens between Shirt and T-shirt.

Section V – Summary and Conclusions

Overall, Fully Connected NN and Convolutional NN both can successfully classify data sets from Fashion MNIST with accuracy at approximately 90%. Achieving accuracy at over 95% is difficult. Efforts are made to investigate the individual relationship between NN hyperparameters and final training results, but due to the large quantity of values involved, it is difficult to fully optimize the system. Nevertheless, the effects of each hyperparameter can be better understood based on the results of this project. The recommended next step is to conduct more experiments with different hyperparameter configurations to quantify the effects of each value to aid future application.

Appendix A – Python functions used

- $partial$: configure hyperparameters for easy usage
- $tf.keras.layers.Sequential$: build NN layers with sequential activation order
- $tf.keras.layers.Flatten$: flatten data as input to NN
- $tf.keras.layers.Dense$: build Fully Connect NN
- $tf.keras.layers.Conv2D$: build 2D convolutional NN
- $model.compile$: compile additional parameters required for model training
- $model.fit$: train model with given hyperparameters and configurations
- $model.predict\_classes$: use trained model for classification
- $confusion\_matrix$: generate confusion matrix for model evaluation

Appendix B – Python codes

```
#%% Initialization
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
import pandas as pd
from sklearn.metrics import confusion_matrix
from functools import partial

#%% Loading
fashion_mnist = tf.keras.datasets.fashion_mnist
(X_train_full, y_train_full), (X_test, y_test) = fashion_mnist.load_data()

#%% Visualization
plt.figure()
for k in range(9):
    plt.subplot(3,3,k+1)
    plt.imshow(X_train_full[k],cmap='gray')
    plt.axis('off')
plt.show
X_train_full.shape

#%% Assign Training and Validation
X_train_full = X_train_full / 255.0 # convert uint8
X_valid = X_train_full[:5000]
X_train = X_train_full[5000:]
X_test = X_test / 255.0
y_valid = y_train_full[:5000]
y_train = y_train_full[5000:]

#%% Setup NN - Fully Connected
# Hyperparameters
my_dense_layer = partial(tf.keras.layers.Dense, activation="relu", kernel_regularizer=t
f.keras.regularizers.l2(0.0001))

model = tf.keras.models.Sequential([
    tf.keras.layers.Flatten(input_shape=[28, 28]),
    my_dense_layer(1000),
    my_dense_layer(10, activation="softmax")
])

# Additional Parameters
model.compile(loss="sparse_categorical_crossentropy",
             optimizer=tf.keras.optimizers.Adam(learning_rate=0.0001),
             metrics=["accuracy"])

# Training NN
history = model.fit(X_train, y_train, epochs=50,validation_data=(X_valid,y_valid))

#%% Loss and Accuracy
pd.DataFrame(history.history).plot(figsize=(8,5))
plt.grid(True)
plt.gca().set_ylim(0,1)
plt.show()

#%% Confusion Matrix
y_pred = model.predict_classes(X_train)
conf_train = confusion_matrix(y_train, y_pred)
print(conf_train)

#%% Implementation on Test Data
model.evaluate(X_test,y_test)
y_pred = model.predict_classes(X_test)
conf_test = confusion_matrix(y_test, y_pred)
print(conf_test)

#%% Output Confusion Matrix
fig, ax = plt.subplots()

# hide axes
fig.patch.set_visible(False)
ax.axis('off')
ax.axis('tight')

# create table and save to file
df = pd.DataFrame(conf_test)
ax.table(cellText=df.values, rowLabels=np.arange(10), colLabels=np.arange(10), loc='cen
ter', cellLoc='center')
fig.tight_layout()
plt.savefig('conf_mat_fc.png',dpi=300)









#%% Assign Training and Validation
```

```
X_train_full = X_train_full / 255.0 # convert uint8
X_valid = X_train_full[:5000]
X_train = X_train_full[5000:]
X_test = X_test / 255.0
y_valid = y_train_full[:5000]
y_train = y_train_full[5000:]

X_valid = X_valid[..., np.newaxis]
X_train = X_train[..., np.newaxis]
X_test = X_test[..., np.newaxis]
#%% Setup NN - Convolutional
# Hyperparameters
my_dense_layer = partial(tf.keras.layers.Dense, activation="tanh", kernel_regularizer=t
f.keras.regularizers.l2(0.0001))
my_conv_layer = partial(tf.keras.layers.Conv2D, activation="tanh", padding="valid")

model = tf.keras.models.Sequential([
    my_conv_layer(20,5,padding="same",input_shape=[28,28,1]),
    tf.keras.layers.AveragePooling2D(2),
    my_conv_layer(10,5),
    tf.keras.layers.AveragePooling2D(2),
    my_conv_layer(50,5),
    tf.keras.layers.Flatten(),
    my_dense_layer(100),
    my_dense_layer(10, activation="softmax")
])

# Additional Parameters
model.compile(loss="sparse_categorical_crossentropy",
             optimizer=tf.keras.optimizers.Adam(learning_rate=0.0001),
             metrics=["accuracy"])

# Training NN
history = model.fit(X_train, y_train, epochs=50, validation_data=(X_valid,y_valid))

#%% Loss and Accuracy
pd.DataFrame(history.history).plot(figsize=(8,5))
plt.grid(True)
plt.gca().set_ylim(0,1)
plt.show()

#%% Confusion Matrix
y_pred = model.predict_classes(X_train)
conf_train = confusion_matrix(y_train, y_pred)
print(conf_train)
```

```python
#%% Implementation on Test Data
model.evaluate(X_test,y_test)
y_pred = model.predict_classes(X_test)
conf_test = confusion_matrix(y_test, y_pred)
print(conf_test)

#%% Output Confusion Matrix
fig, ax = plt.subplots()

# hide axes
fig.patch.set_visible(False)
ax.axis('off')
ax.axis('tight')

# create table and save to file
df = pd.DataFrame(conf_test)
ax.table(cellText=df.values, rowLabels=np.arange(10), colLabels=np.arange(10), loc='cen
ter', cellLoc='center')
fig.tight_layout()
plt.savefig('conf_mat_cnn.png',dpi=300)
```

Appendix C – Additional Figures during Training

Fully Connected NN

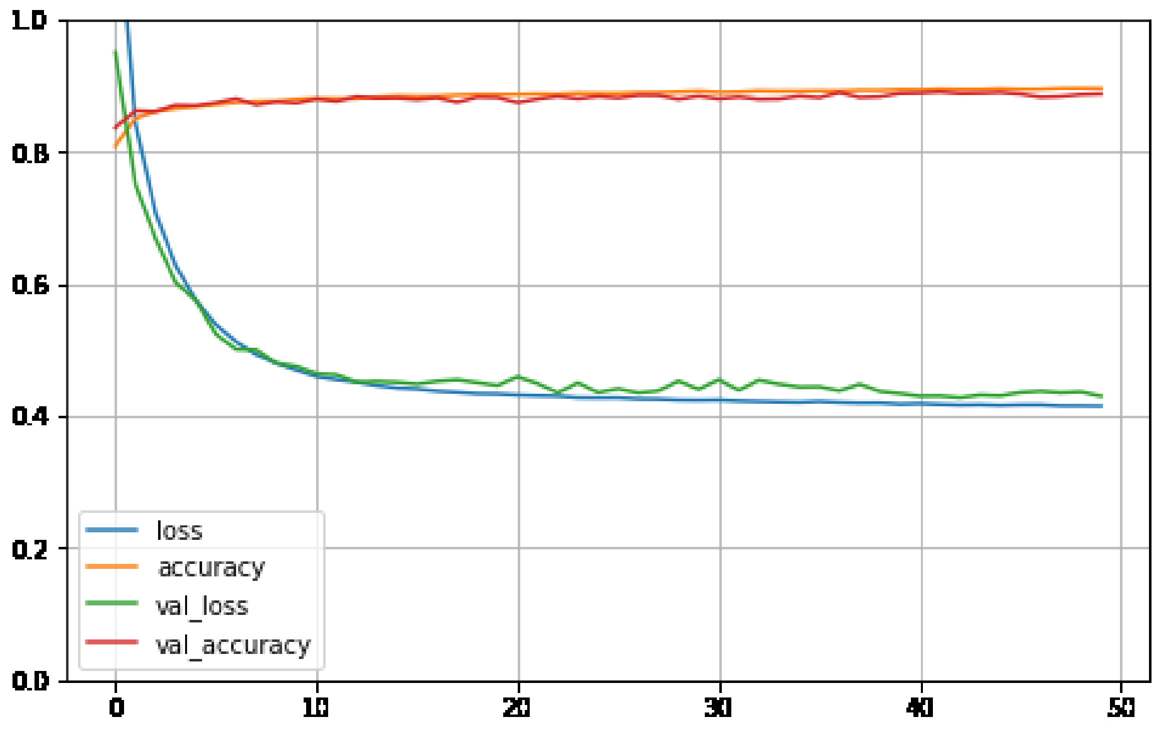


Fig. C1 Regularizer set to 0.001

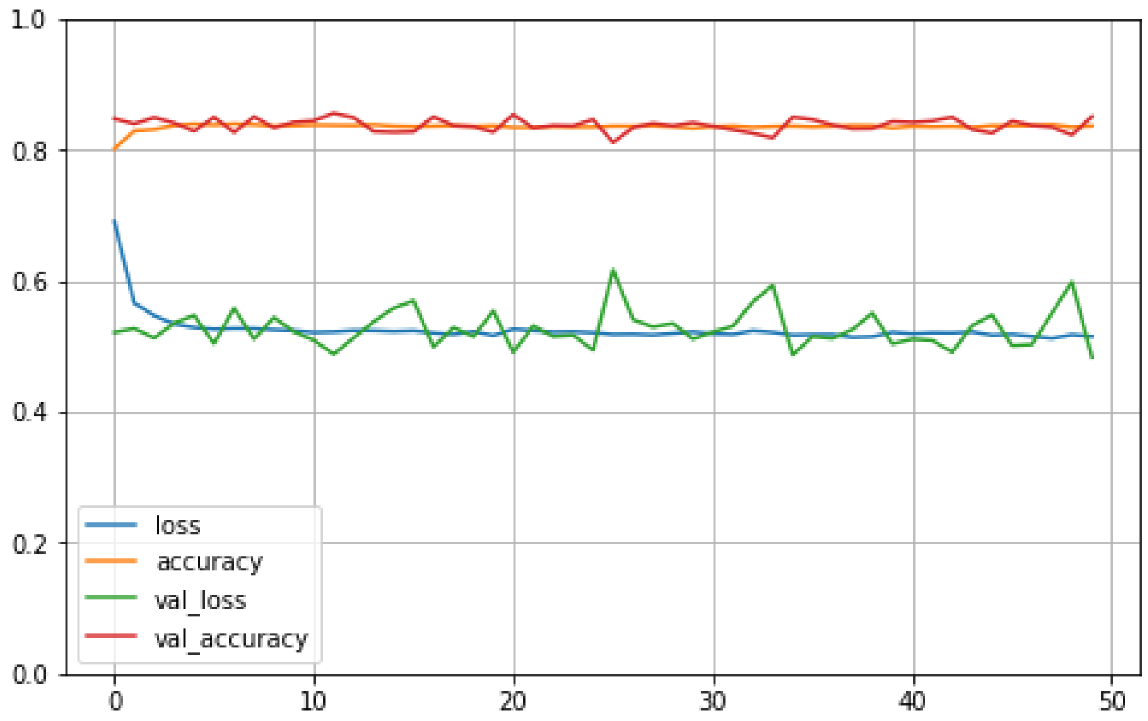


Fig. C2 Learning rate set to 0.01

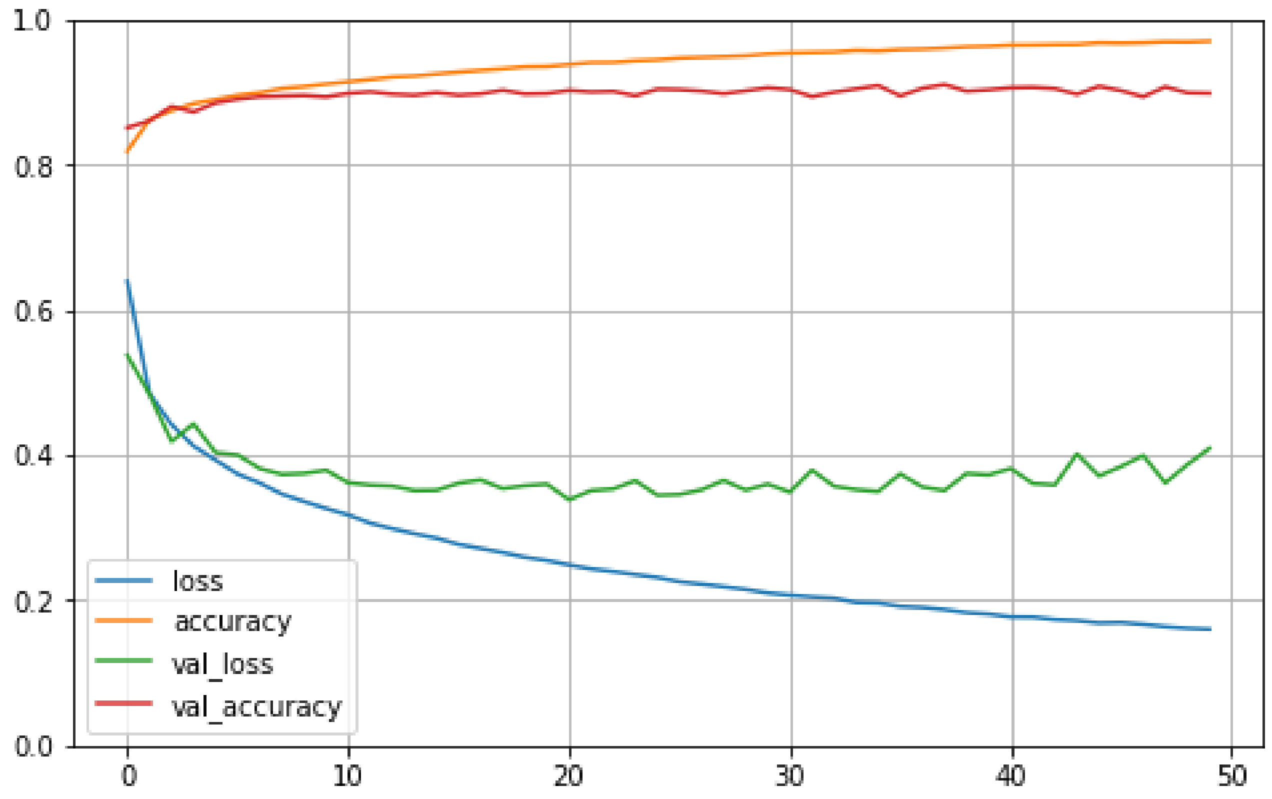


Fig. C3 Layer configuration 1000; 100; 10

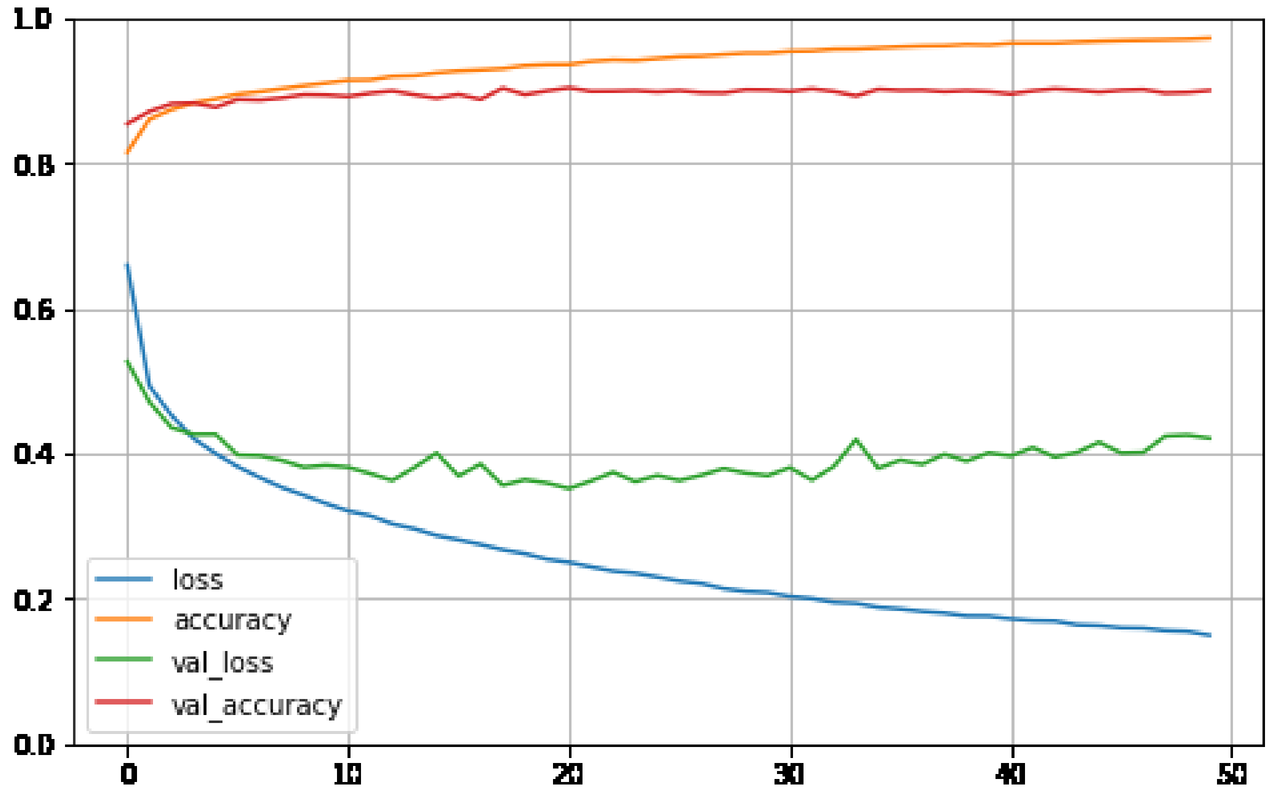


Fig. C4 Layer configuration 1000; 100; 100; 10

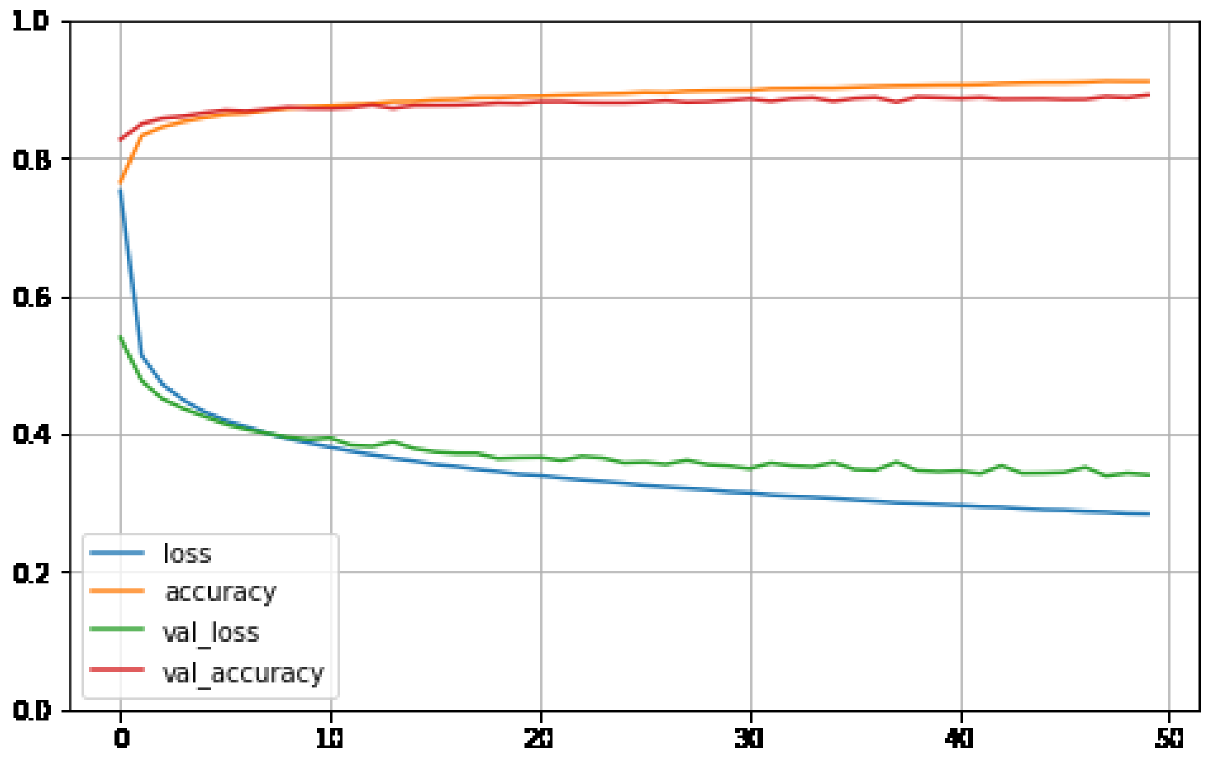


Fig. C5 Layer configuration 100; 10

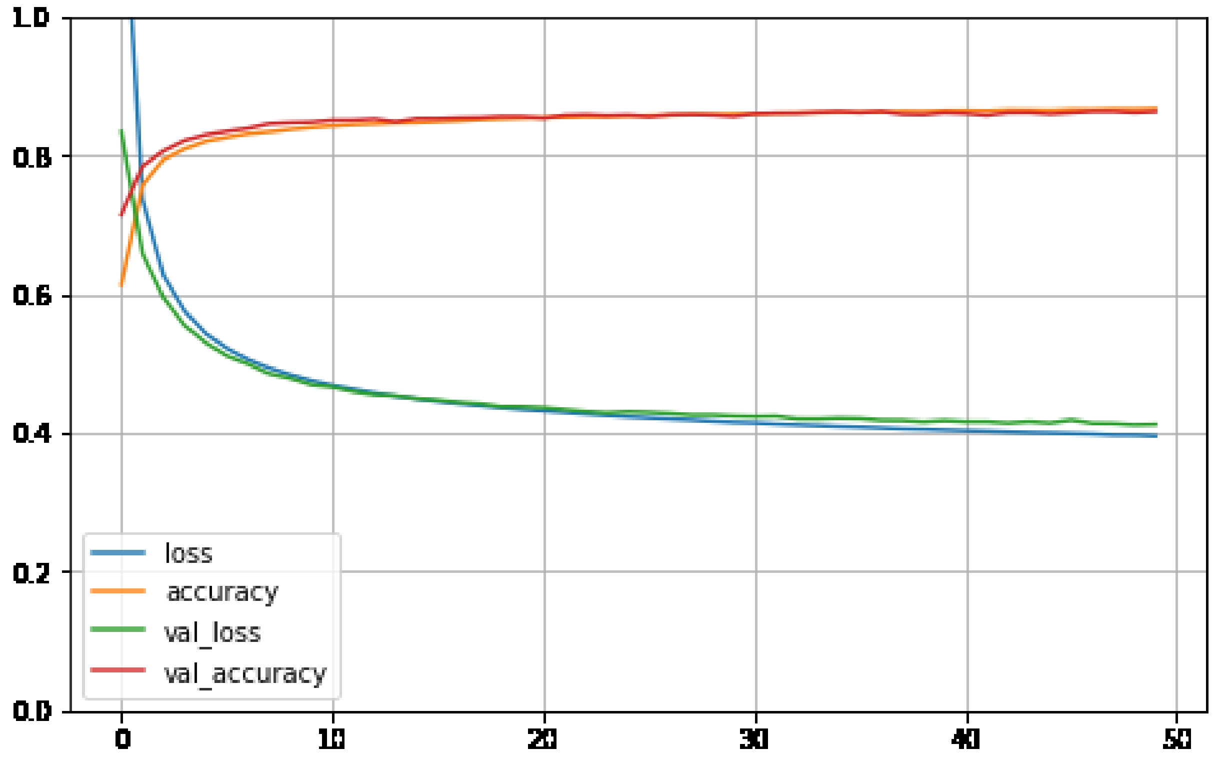


Fig. C6 Layer configuration 10; 10

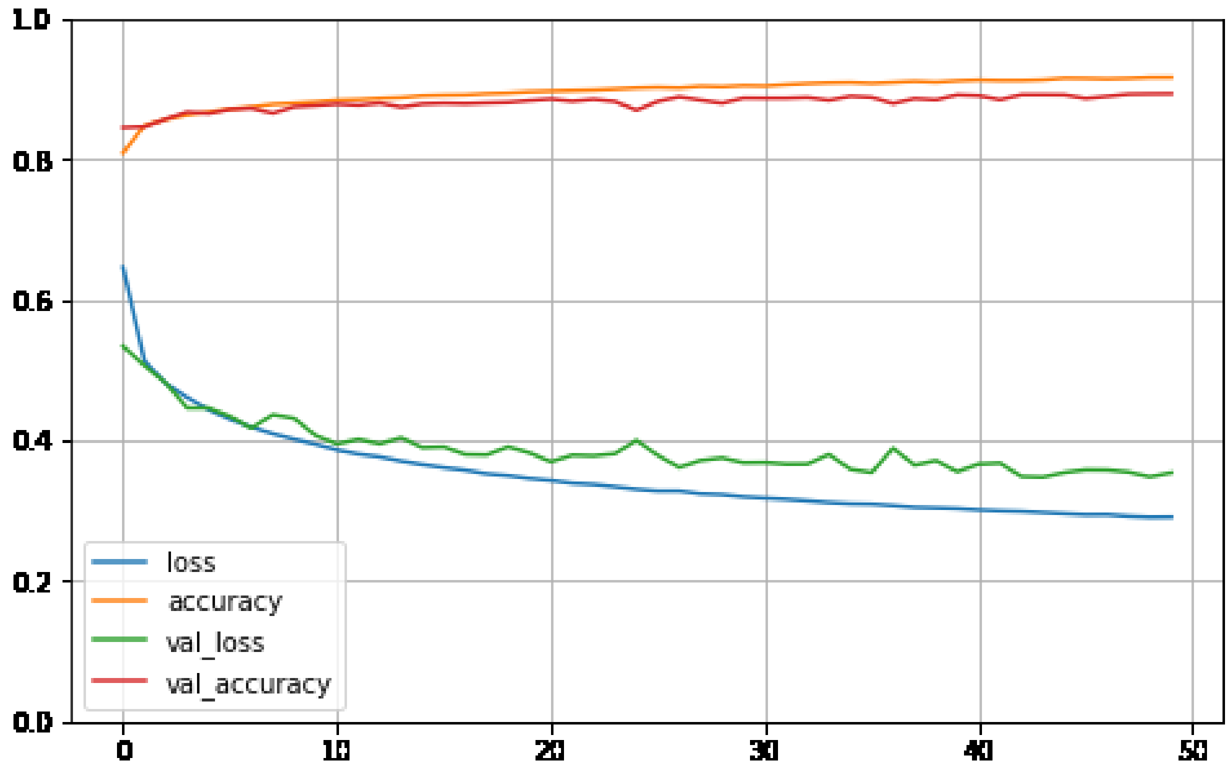


Fig. C7 Activation function set to $tanh$

Convolutional NN

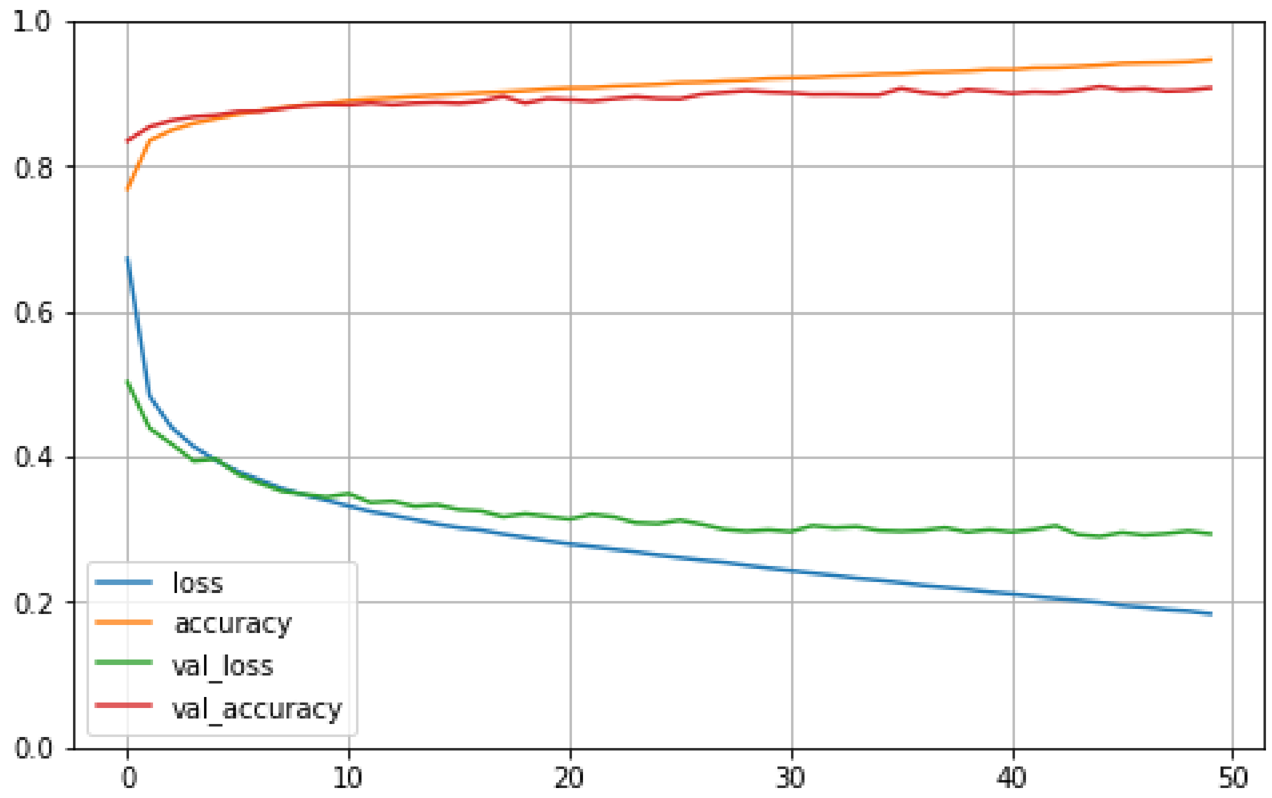


Fig. C8 $LeNet$ with all convolutional layers “same” padding

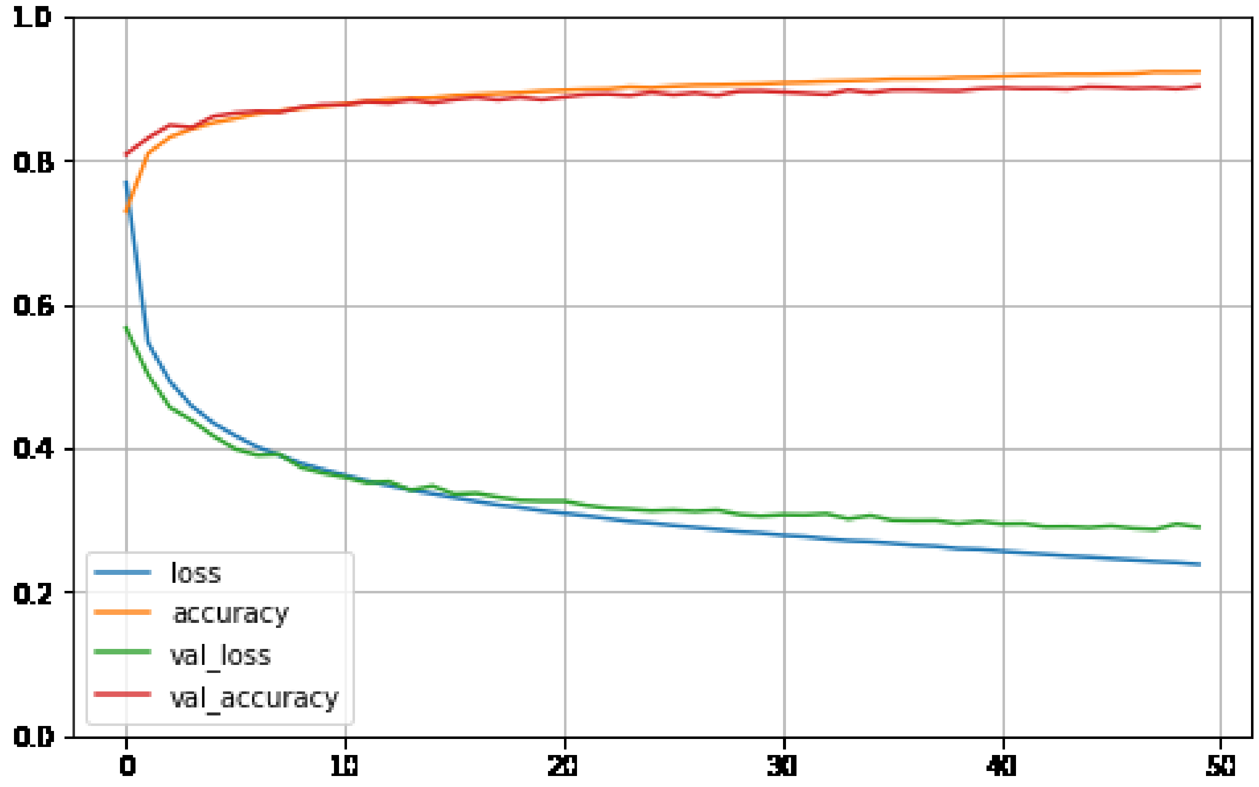


Fig. C9 $LeNet$ with kernel size 3

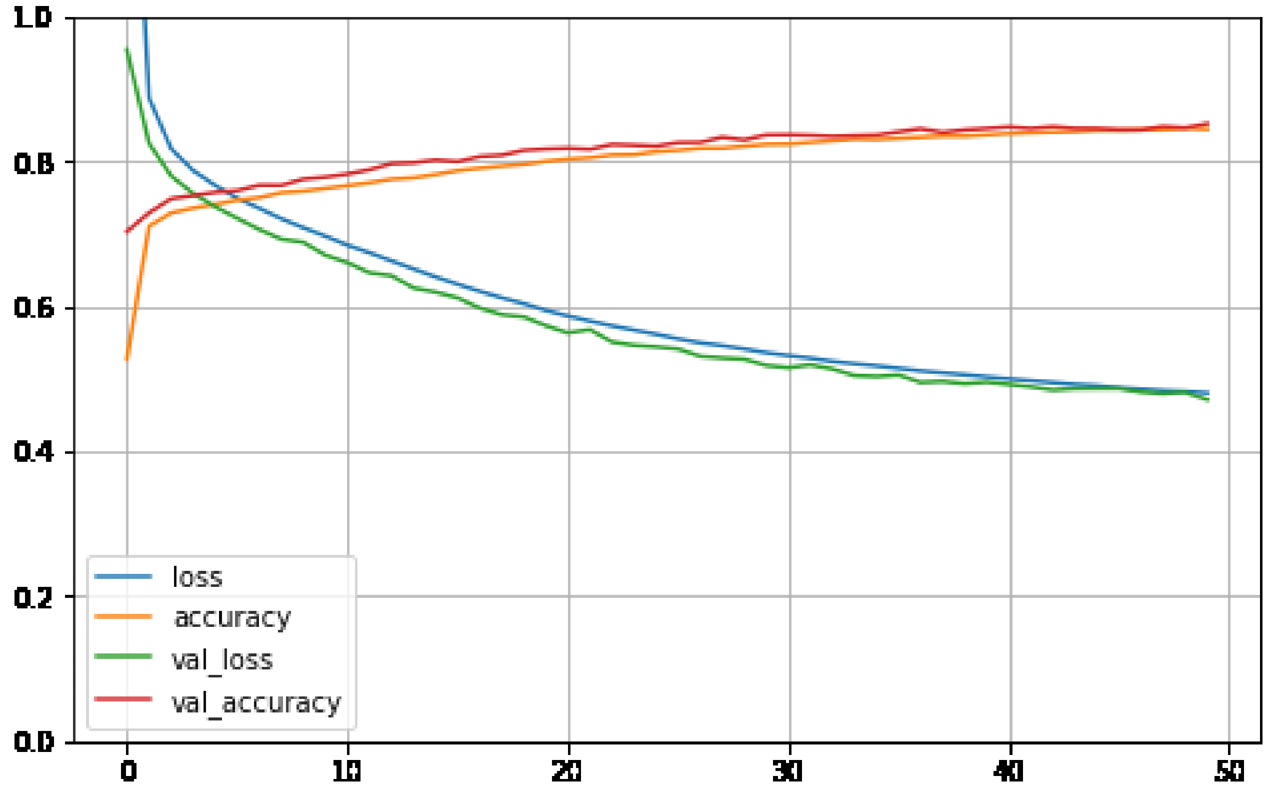


Fig. C10 $LeNet$ with all kernel size 3, last convolutional layer with stride 2

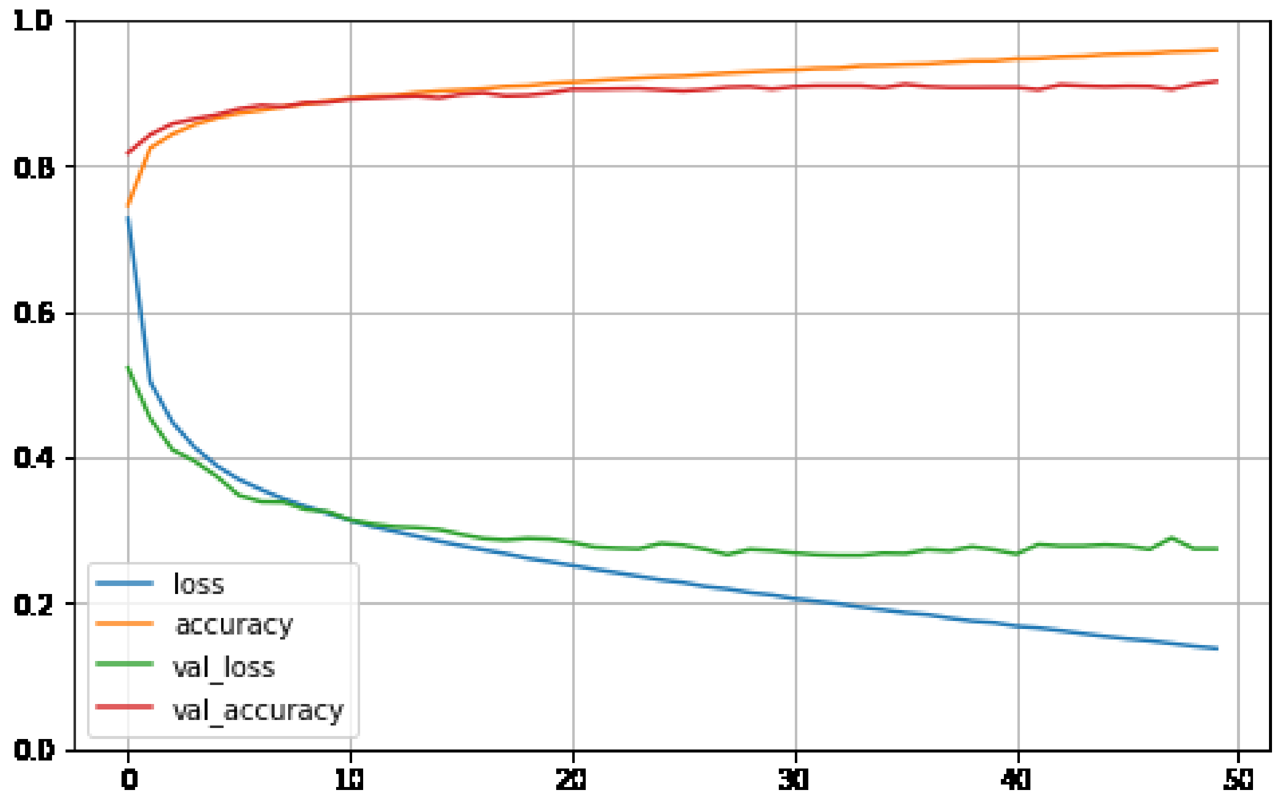


Fig. C11 $LeNet$ 20, 30 and 200 features in the convolutional layers