223142768_SIT744_assignment1_solution

August 17, 2024

1 SIT744 Assignment 1

Run the following code before you start.

[1]:
```
import getpass
import datetime
import socket

def generate_author_claim():
    # Get current user
    user = getpass.getuser()

    # Get current timestamp
    timestamp = datetime.datetime.now().strftime("%Y-%m-%d %H:%M:%S")

    # Get current IP address
    ip_address = socket.gethostbyname(socket.gethostname())

    # Enter your name
    name = input("Enter your full name: ")

    # Enter your email address
    email = input("Enter your email address: ")

    # Enter your student ID
    student_id = input("Enter your student ID: ")

    # Generate author claim string
    author_claim = f"Code authored by {user} ({name} {student_id} {email}) on␣
 ↪{timestamp}  from IP address {ip_address}"

    return author_claim

# Generate the author claim string
author_claim = generate_author_claim()

# Print the author claim string
print(author_claim)
```

```
Enter your full name: Huu Phuc Hong
Enter your email address: s223142768@deakin.edu.au
Enter your student ID: 223142768
Code authored by root (Huu Phuc Hong 223142768 s223142768@deakin.edu.au) on
2024-08-17 04:24:35  from IP address 172.28.0.12
```

1.1 Set 1 (P Tasks) Construct a forward neural network

(weight ~60%)

With this set of tasks, you are going to build a feedforward neural network for a classification task. You will train the model on the Fashion MNIST dataset (https://github.com/zalandoresearch/fashion-mnist).

1.1.1 Task 1.1 Understanding the data

(weight ~20%)

1. Describe the Fashion-MNIST dataset. How many classes exist? Display 5 training examples from each target class. Do you see any patterns?

Answer:

- The Fashion-MNIST dataset contains total 70000 instances with labels, in which the training dataset contains 60000 instances, and the test dataset contains 10000 instances.
- Each instance in the dataset is represented as a 28x28 pixel square image, meaning both the width and height of each image are 28 pixels.
- Each image is in grayscale, with pixel values ranging from 0 to 255.
- The dataset has 10 classes in total, each represents a unique category as below.

Label	Category
0	T-shirt/top
1	Trouser
2	Pullover
3	Dress
4	Coat
5	Sandal
6	Shirt
7	Sneaker
8	Bag
9	Ankle boot

- Based on the examples from each class, I observe that each class contains different instances with varying shapes. However, some categories appear quite similar due to their grayscale color and overall appearance, such as Pullover and Shirt. On the other hand, there are categories that are distinctly different, such as Bag compared to T-shirt or Trouser. My first guess is that the model will find it relatively easy to distinguish between categories like Bag and Trouser, but it might struggle more with categories like Pullover and Shirt due to their visual similarities.

[ ]:
```
%pip install umap_learn --quiet
```

[ ]:
```
%pip install tensorflow==2.15.0 --quiet
```

[4]:
```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import sklearn
from sklearn.manifold import TSNE
from sklearn.model_selection import train_test_split
import tensorflow as tf
from tensorflow import keras
from keras.callbacks import TensorBoard, LearningRateScheduler
from keras.optimizers.schedules import ExponentialDecay
from keras.initializers import RandomUniform
from keras.models import Model
from keras.utils import plot_model, to_categorical
from keras import layers, Sequential
from keras.datasets import fashion_mnist
from keras.regularizers import l2
import umap
from datetime import datetime
import time
import os
```

[5]:
```
# Version of packages
print(f"NumPy version: {np.__version__}")
print(f"Seaborn version: {sns.__version__}")
print(f"scikit-learn version: {sklearn.__version__}")
print(f"TensorFlow version: {tf.__version__}")
print(f"UMAP version: {umap.__version__}")
```

```
NumPy version: 1.26.4
Seaborn version: 0.13.1
scikit-learn version: 1.3.2
TensorFlow version: 2.15.0
UMAP version: 0.5.6
```

[6]:
```
# Load fashion mnist
(train_images, train_labels), (test_images, test_labels) = fashion_mnist.
↪load_data()
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-
datasets/train-labels-idx1-ubyte.gz
29515/29515 [==============================] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-
datasets/train-images-idx3-ubyte.gz
26421880/26421880 [==============================] - 0s 0us/step
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-
datasets/t10k-labels-idx1-ubyte.gz
5148/5148 [==============================] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-
datasets/t10k-images-idx3-ubyte.gz
4422102/4422102 [==============================] - 0s 0us/step
```

[7]:
```
# Check the overall shape of the loaded dataset
print("Shape of train dataset:", train_images.shape)
print("Shape of test dataset:", test_images.shape)
```

```
Shape of train dataset: (60000, 28, 28)
Shape of test dataset: (10000, 28, 28)
```

[8]:
```
# Check number of instances of the loaded dataset
print("Number of instances in train dataset:", train_images.shape[0])
print("Number of instances in test dataset:", test_images.shape[0])
```

```
Number of instances in train dataset: 60000
Number of instances in test dataset: 10000
```

[9]:
```
# Check width/height of the loaded dataset
print("Width of train dataset:", train_images.shape[2])
print("Height of train dataset:", train_images.shape[1])
print("Width of test dataset:", test_images.shape[2])
print("Height of test dataset:", test_images.shape[1])
```

```
Width of train dataset: 28
Height of train dataset: 28
Width of test dataset: 28
Height of test dataset: 28
```

[10]:
```
# Check number of classes of the loaded dataset
print("Number of classes in train dataset:",len(np.unique(train_labels)))
print("Number of classes in test dataset:",len(np.unique(test_labels)))
```

```
Number of classes in train dataset: 10
Number of classes in test dataset: 10
```

[11]:
```
# Check the minimum/maximum color scale of the loaded dataset
print("Minimum pixel value:", np.min(train_images))
print("Maximum pixel value:", np.max(train_images))
```

```
Minimum pixel value: 0
Maximum pixel value: 255
```

[12]:
```
# Create class name according to numeric label order
class_names = ["T-shirt/top", "Trouser", "Pullover", "Dress", "Coat", "Sandal",␣
 ↪"Shirt", "Sneaker", "Bag", "Ankle boot"]
```

```python
# Set up the figure and axes (subplot with 5 rows, 10 columns)
fig, axes = plt.subplots(5, 10, figsize=(15, 15))

# Loop through each class to display instance of that class
for i in range(10):
    class_images = train_images[train_labels == i]
    # Display just 5 instances for each class
    for j in range(5):
        axes[j, i].imshow(class_images[j], cmap="gray")
        axes[j, i].axis("off")
        # Display title on top of images
        if j == 0:
            axes[j, i].set_title(class_names[i])

# Display the figure
plt.show()
```

2. Prepare the data for learning a neural network, including creating training, validation, and test datasets. What preprocessing steps are required and why? How many training examples and how many test examples are you using?

Answer:

- I have training dataset with 54000 instances, validation dataset with 6000 instances and test dataset with 10000 instance.
- Preprocessing steps:
  - Reshape the data to 1D Numpy array because I will use fully connected networks for this assignment.
  - Normalize data to [0, 1] range as this is a good practice and this helps the network learn faster. Without normalization, larger values could dominate the learning process, leading to suboptimal model.
  - Apply one-hot encoding as this is needed for categorical variable.

[13]:
```
# Reshape the images into 1D Numpy array
train_images = train_images.reshape(train_images.shape[0], 28*28)
test_images = test_images.reshape(test_images.shape[0], 28*28)
```

[14]:
```
# Normalize input data to be between 0 and 1
train_images = train_images.astype("float32") / 255
test_images = test_images.astype("float32") / 255
```

[15]:
```
# Apply one-hot encoder to the labels
train_labels = to_categorical(train_labels)
test_labels = to_categorical(test_labels)
```

[16]:
```
# Split the training set (100%) into training (90%) and validation sets (10%)
train_images, val_images, train_labels, val_labels =␣
 ↪train_test_split(train_images, train_labels, test_size=0.1, random_state=42)

# Check number of instances in each train/validate/test dataset
print(f"Number of training examples: {train_images.shape[0]}")
print(f"Number of validation examples: {val_images.shape[0]}")
print(f"Number of test examples: {test_images.shape[0]}")
```

```
Number of training examples: 54000
Number of validation examples: 6000
Number of test examples: 10000
```

1.1.2 Task 1.2 Setting up a model for training

(weight ~ 20%)

Construct a deep feedforward neural network. In other words, you can use **only fully connected (dense) layers**. You need to decide and report the following configurations:

- Output layer:

  - How many output nodes?
  - Which activation function?
- Hidden layers:
  - How many hidden layers?
  - How many nodes in each layer?
  - Which activation function for each layer?
- Input layer
  - What is the input size?
  - Do you need to reshape the input? Why?

Justify your model design decisions.

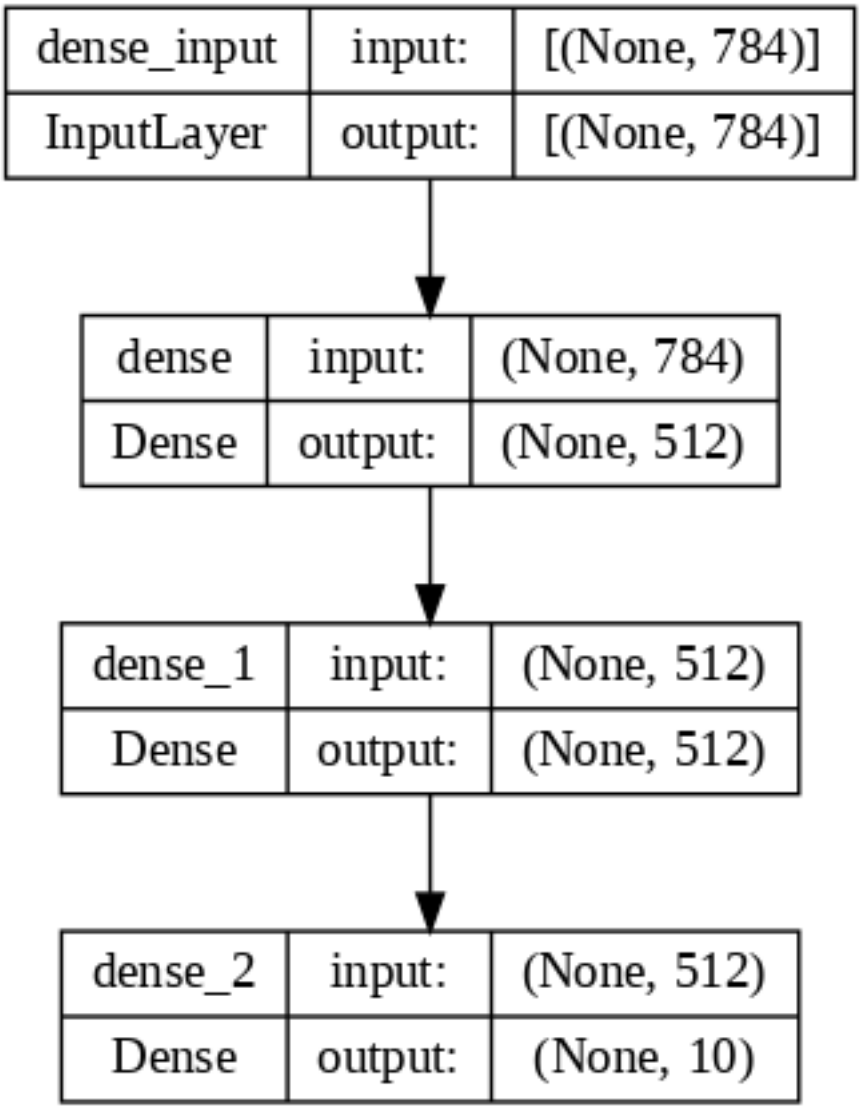
Plot the model structure using `keras.utils.plot_model` or similar tools.

Answer:

- Output layer:
  - 10 output nodes as there are 10 classes in total (one for each class).
  - I would use softmax activation function because this is a multi-class classification problem. This activation function will calculate the probabilities for each class, sum of which is 100%.
- Hidden layers:
  - I would begin with 2 hidden layers to balance convergence speed and complexity of the model. With low number of hidden layers, it will reduce the complexity of model and reduce overfitting.
  - Each layer contains 512 nodes. I think this is a suitable starting point to balance the complexity of model.
  - I choose ReLU activation function for each layer because its non-linearity helps the model learn complex patterns in the data. It also mitigrates issues like vanishing gradients effectively.
- Input layer:
  - Input size is 28x28 pixels (784 after reshaping)
  - We need to reshape the input because 1D Numpy array is the input requirement for fully connected network.

[17]:
```
# Configure layers' number of neurons, activation functions
model = Sequential([layers.Dense(512, activation="relu", input_shape=(28*28,)),
                    layers.Dense(512, activation="relu"),
                    layers.Dense(10, activation="softmax")])

# Display configured model
plot_model(model, show_shapes=True, dpi = 100, show_layer_names=True)
```

[17]:



1.1.3 Task 1.3 Fitting the model

(weight ~ 20%)

Decide and report the following settings:

- The loss function
- The metrics for model evaluation (which may be different from the loss function)

Explain their roles in model fitting.

Decide the optimiser that you will use. Also report the following settings:

1. The training batch size
2. The number of training epochs

3. The learning rate. If you used momentum or a learning rate schedule, please report the configuration as well.

Justify your decisions.

Now fit the model. Show how the training loss and the evaluation metric change. How did you decide when to stop training?

Answer:

- Loss function: I choose "categorical crossentropy" because this is a multi-class classification problem. It measures the difference between the true labels (already apply one-hot encoder) and the predicted probabilities. It will help minimize error for this task by measuring how good the algorithm is doing.
- Metrics for model evaluation: I choose "accuracy" because it is a straightforward metric to measure the model performance - percentage of correct classified images.
- Oprimizer: I choose "Adam" optimizer because it is typically used, helping in faster convergence and better handling of sparse gradients.
- Training batch size: 128
- Training epoch: 20
- We can plot the training & validation accuracy/loss (included in TensorBoard) to see from which epoch, the model will start overfitting (if the validation loss/accuracy starts to fluctuate (increase/decrease) and occurs the noticeable gap between training accuracy and validation accuracy). Then we can stop training from that epoch. However, I won't stop the training in this assignment so that we can notice where and how overfitting occurs later.

[18]:
```
# Configure model's optimizer, loss function and metrics
model.compile(optimizer="Adam",
              loss="categorical_crossentropy",
              metrics=["accuracy"])
```

[19]:
```
# This is used for task 2.1C TensorBoard
logdir = os.path.join("logs/final", datetime.now().strftime("%Y%m%d-%H%M%S"))
tb_callback = TensorBoard(log_dir=logdir, histogram_freq=1)
```

[20]:
```
# Fit train/validate dataset to configured model
# I choose verbose = 0 so that the training logs don't appear
# because there is a requirement to clean up the code outputs to reduce
# unnecessary information (e.g., excessively long training logs)
history = model.fit(train_images,
                    train_labels,
                    epochs=20,
                    batch_size=128,
                    validation_data=(val_images, val_labels),
                    callbacks=[tb_callback],
                    verbose=0)

# Evaluate model's performance
test_loss, test_accuracy = model.evaluate(test_images, test_labels, verbose=0)
```

```
print("Test Loss:", test_loss)
print("Test Accuracy:", test_accuracy)
```

```
Test Loss: 0.3690487742424011
Test Accuracy: 0.8934000134468079
```

1.2 Set 2 (C Tasks) Improve the model

(weight ~ 10%)

1.2.1 Task 2.1 Check the training using visualisation

Visualise the training process (e.g., using TensorBoard). Show screenshots of visualisation.

Do you see overfitting or underfitting? Why?

Answer:

- From the epoch accuracy and loss visualizations in TensorBoard, it shows that both accuracy/loss start to decrease/increase after the first three epochs (step 0, 1, 2). The model begins to overfit after these initial epochs as the training accuracy/loss keep increasing/decreasing respectively while the validation accuracy/loss fluctuates. This issue might be due to either the high number of epochs or the high model's complexity.

[21]:
```
%reload_ext tensorboard
%tensorboard --logdir=logs/final
```

```
<IPython.core.display.Javascript object>
```

1.2.2 Task 2.2 Apply regularisation

Improve the training process by applying regularisation. Below are some options:

1. Dropout
2. Batch normalisation
3. L2 Regularisation

Compare the effect of different regularisation techniques on model training. You may also try other techniques for improving training such as learning rate scheduling (see https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/schedules/LearningRateSchedule).

Answer

- Case 1 - Using Dropout: This regularization method randomly deactivates a fraction of neurons, as specified by the user, to help prevent overfitting. I have selected a dropout rate of 30% for both the first and second layers. This choice aims to strike a balance between model complexity and simplicity. All other configurations remain unchanged to assess whether applying regularization improves performance for this Fashion MNIST dataset.
- Case 2 - Using Batch Normalization: This method normalizes the inputs of each layer during training, which helps mitigate issues like internal covariate shift. I will apply Batch Normalization to both layers while keeping all other configurations the same.

- Case 3 - Using L2 Regularization: This method adds a penalty proportional to the square of the magnitude of weights to the loss function, which helps prevent overfitting. In this case, I will apply L2 regularization with a regularization strength of 0.01 to the dense layers of the model. All other configurations will remain the same.
- Case 4 - Learning Rate Scheduling: This method adjusts the learning rate during training to help improve model convergence. By gradually reducing the learning rate, it potentially leads to better performance. I will apply a learning rate schedule using exponential decay to the model while keeping all other configurations unchanged.

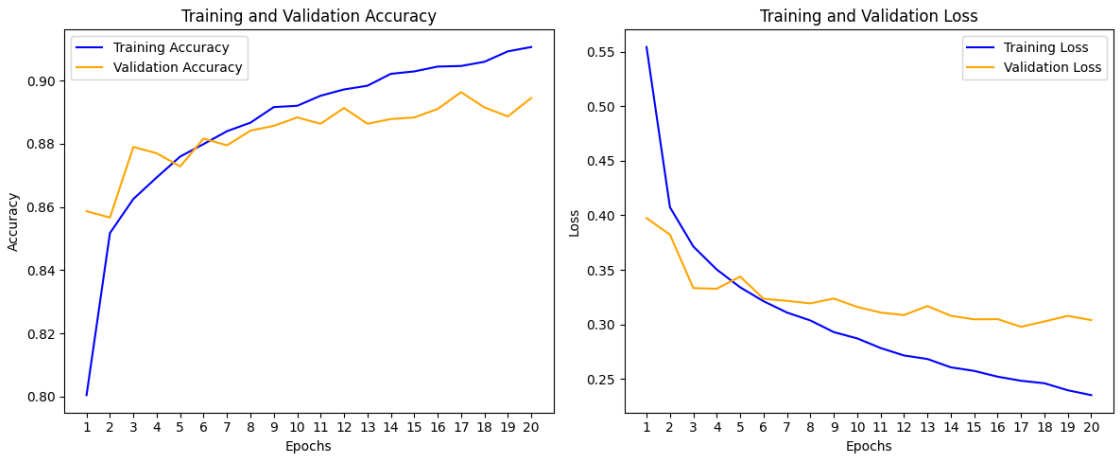
2 Case 1 - Using Dropout

[22]:
```python
# Create plot_train_hist function to check the training process
def plot_train_hist(history):
    # Extract training/validate accuracy and loss
    train_accuracy = history.history["accuracy"]
    val_accuracy = history.history["val_accuracy"]
    train_loss = history.history["loss"]
    val_loss = history.history["val_loss"]

    # Number of epochs
    epochs = range(1, len(train_accuracy) + 1)

    # Plot training/validation accuracy
    plt.figure(figsize=(12, 5))
    plt.subplot(1, 2, 1)
    plt.plot(epochs, train_accuracy, "blue", label="Training Accuracy")
    plt.plot(epochs, val_accuracy, "orange", label="Validation Accuracy")
    plt.title("Training and Validation Accuracy")
    plt.xlabel("Epochs")
    plt.ylabel("Accuracy")
    plt.xticks(epochs)
    plt.legend()

    # Plot training/validation loss
    plt.subplot(1, 2, 2)
    plt.plot(epochs, train_loss, "blue", label="Training Loss")
    plt.plot(epochs, val_loss, "orange", label="Validation Loss")
    plt.title("Training and Validation Loss")
    plt.xlabel("Epochs")
    plt.ylabel("Loss")
    plt.xticks(epochs)
    plt.legend()

    plt.tight_layout()
    plt.show()
```

[23]:
```
# Configure layers' number of neurons, activation functions, add Dropout␣
 ↪regularizer technique
model_dropout = Sequential([layers.Dense(512, activation="relu",␣
 ↪input_shape=(28*28,)),
                            layers.Dropout(0.3),
                            layers.Dense(512, activation="relu"),
                            layers.Dropout(0.3),
                            layers.Dense(10, activation="softmax")])

# Configure model's optimizer, loss function and metrics
model_dropout.compile(optimizer="Adam",
                      loss="categorical_crossentropy",
                      metrics=["accuracy"])

# Fit train/validate dataset to configured model
history = model_dropout.fit(train_images,
                            train_labels,
                            epochs=20,
                            batch_size=128,
                            validation_data=(val_images, val_labels),
                            verbose=0)

# Evaluate model's performance
test_loss, test_accuracy = model_dropout.evaluate(test_images, test_labels,␣
 ↪verbose=0)

print("Test Loss:", test_loss)
print("Test Accuracy:", test_accuracy)
```

```
Test Loss: 0.3169795870780945
Test Accuracy: 0.8898000121116638
```

[24]:
```
plot_train_hist(history)
```



3 Case 2 - Using Batch Normalization

[25]:
```
# Configure layers' number of neurons, add BatchNormalization regularizer␣
 ↪technique
# before configuring activation function
model_bn = Sequential([layers.Dense(512, input_shape=(28*28,)),
                       layers.BatchNormalization(),
                       layers.Activation("relu"),
                       layers.Dense(512),
                       layers.BatchNormalization(),
                       layers.Activation("relu"),
                       layers.Dense(10, activation="softmax")])

# Configure model's optimizer, loss function and metrics
model_bn.compile(optimizer="Adam",
                 loss="categorical_crossentropy",
                 metrics=["accuracy"])

# Fit train/validate dataset to configured model
history = model_bn.fit(train_images,
                       train_labels,
                       epochs=20,
                       batch_size=128,
                       validation_data=(val_images, val_labels),
                       verbose=0)

# Evaluate model's performance
test_loss, test_accuracy = model_bn.evaluate(test_images, test_labels,␣
 ↪verbose=0)

print("Test Loss:", test_loss)
print("Test Accuracy:", test_accuracy)
```

```
Test Loss: 0.47556304931640625
Test Accuracy: 0.8824999928474426
```

[26]:
```
plot_train_hist(history)
```

4 Case 3 - Using L2 Regularization

[27]:
```
# Configure layers' number of neurons, activation functions, add L2 regularizer␣
 ↪technique
model_l2 = Sequential([layers.Dense(512, activation="relu",␣
 ↪input_shape=(28*28,), kernel_regularizer=l2(0.01)),
                       layers.Dense(512, activation="relu",␣
 ↪kernel_regularizer=l2(0.01)),
                       layers.Dense(10, activation="softmax")])

# Configure model's optimizer, loss function and metrics
model_l2.compile(optimizer="Adam",
                 loss="categorical_crossentropy",
                 metrics=["accuracy"])

# Fit train/validate dataset to configured model
history = model_l2.fit(train_images,
                       train_labels,
                       epochs=20,
                       batch_size=128,
                       validation_data=(val_images, val_labels),
                       verbose=0)

# Evaluate model's performance
test_loss, test_accuracy = model_l2.evaluate(test_images, test_labels,␣
 ↪verbose=0)

print("Test Loss:", test_loss)
print("Test Accuracy:", test_accuracy)
```

```
Test Loss: 0.5129541754722595
Test Accuracy: 0.845300018787384
```

[28]:
```
plot_train_hist(history)
```

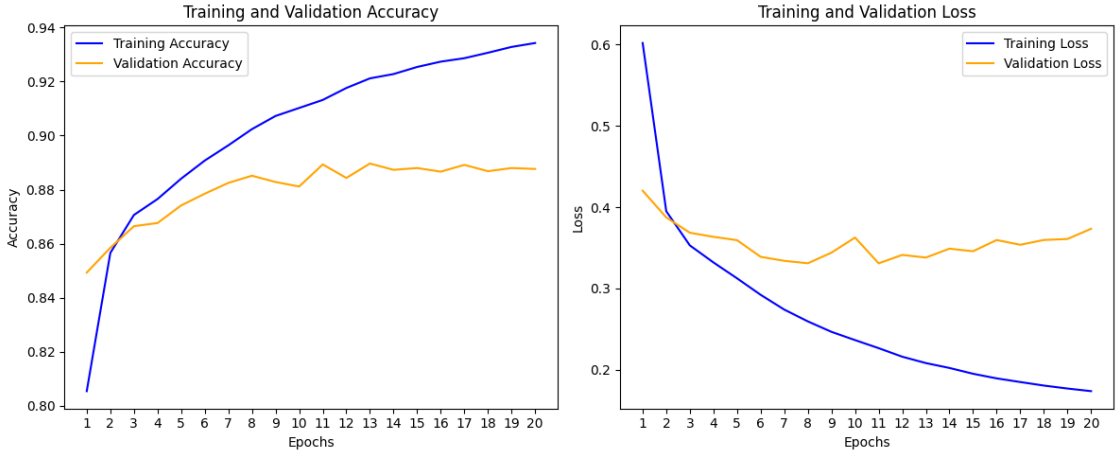
5 Case 4 - Learning Rate Scheduling

[29]:
```
# Configure learning rate scheduling
lr_schedule = ExponentialDecay(initial_learning_rate = 0.01,
                               decay_steps=100,
                               decay_rate=0.96,
                               staircase=True)

# Configure learning rate for Adam optimizer
optimizer = keras.optimizers.Adam(learning_rate=lr_schedule)

# Configure layers' number of neurons, activation functions
model_lrs = Sequential([layers.Dense(512, activation="relu",␣
 ↪input_shape=(28*28,)),
                        layers.Dense(512, activation="relu"),
                        layers.Dense(10, activation="softmax")])

# Configure model's optimizer, loss function and metrics
model_lrs.compile(optimizer=optimizer,
                  loss="categorical_crossentropy",
                  metrics=["accuracy"])

# Fit train/validate dataset to configured model
history = model_lrs.fit(train_images,
                        train_labels,
```

```
                        epochs=20,
                        batch_size=128,
                        validation_data=(val_images, val_labels),
                        callbacks=[LearningRateScheduler(lr_schedule)],
                        verbose=0)

# Evaluate model's performance
test_loss, test_accuracy = model_lrs.evaluate(test_images, test_labels,␣
 ↪verbose=0)

print("Test Loss:", test_loss)
print("Test Accuracy:", test_accuracy)
```

```
Test Loss: 0.38670840859413147
Test Accuracy: 0.8840000033378601
```

[30]:
```
plot_train_hist(history)
```



6 Summary

Model	Test Loss	Test Accuracy
Base	0.369	0.893
Base + Dropout	0.317	0.890
Base + Batch Normalization	0.476	0.882
Base + L2 Regularizer	0.513	0.845
Base + LRS	0.387	0.884

- The base model achieved the best performance with 89.3% accuracy and a test loss of 0.369.
- Adding dropout improved the test loss to 0.317, suggesting better generalization, though accuracy slightly decreased to 89% => it helped optimize the learning process without sig-

nificantly impacting accuracy. As we can see from the accuracy/loss plot, this regularizer is really effective in reducing overfitting for training this dataset as the gap between training and validate accuracy/loss is reduced compared to the base model.

- Batch normalization increased the test loss to 0.476 and reduced accuracy to 88.2%. Also, the overfitting problem as we can see from the accuracy/loss plot seems to be worse => it might not be as effective for this model.
- The L2 regularizer led to the highest test loss at 0.513 and the lowest accuracy of 84.5%. The plot also show that this regularizer is very effective in solving overfitting problem, suggesting possible tuning necessity for better accuracy.
- Learning rate scheduling (LRS) increased the test loss to 0.387 and decreasing an accuracy to 88.4%. Also, it don't really have any effect in reducing overfitting problem in this case, suggesting possible tuning necessity.

=> Adding dropout is the best regularizer chosen to optimize the training process.

=> However, as the accuracy acquired from base model is higher, I will choose it for the following visualization tasks below.

6.0.1 Task 2.3 Visualise the trained network

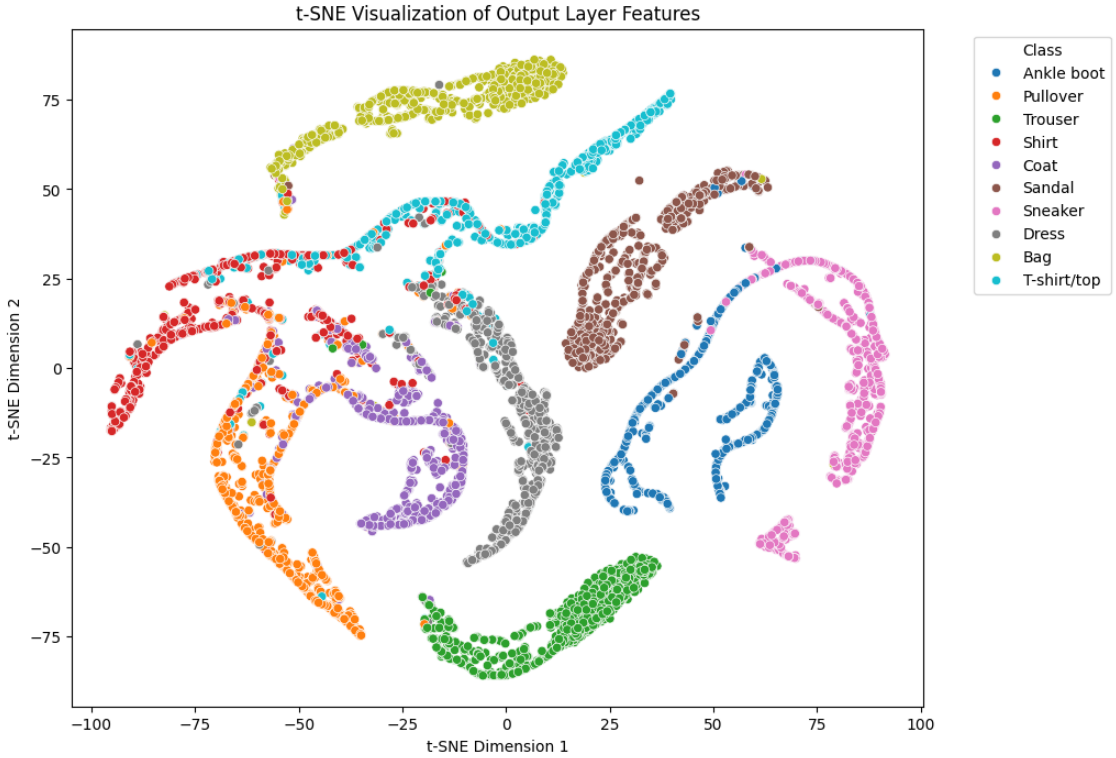
Once the network is trained, extract the output layer"s features for a set of test data points. Use the t-SNE algorithm to reduce the dimensionality of the extracted output features to 2D or 3D for visualization. Create a scatter plot to visualize the t-SNE results. Each point in the plot should represent a data sample, and colors should represent the true class labels. What do you observe?

Answer:

- The plot shows somewhat clear distinct clusters of data points, each representing different true class labels. This indicates that the neural network has learned to some extent to differentiate between the classes (especially for Sandal, Bag, Trouser).
- While there are distinct clusters, some classes are closer to each other or have overlapping regions. This indicates that there might be similarities in the features of these classes that the model hasn't been able to learn effectively, making them harder to distinguish (especially for Coat, Shirt, Pullover, T-shirt).

[31]:
```
# Predict labels using base model (model used in Pass task)
test_features = model.predict(test_images, verbose=0)

# Apply t-SNE to reduce dimensionality
tsne = TSNE(n_components=2, random_state=42)
test_features_tsne = tsne.fit_transform(test_features)

# Get predicted class labels
predicted_classes = np.argmax(test_labels, axis=1)

# Get class names
predicted_class_names = [class_names[i] for i in predicted_classes]

# Create a scatter plot
plt.figure(figsize=(10, 8))
```

```
sns.scatterplot(x=test_features_tsne[:, 0], y=test_features_tsne[:, 1],␣
 ↪hue=predicted_class_names, palette="tab10")
plt.title("t-SNE Visualization of Output Layer Features")
plt.xlabel("t-SNE Dimension 1")
plt.ylabel("t-SNE Dimension 2")
plt.legend(title="Class", bbox_to_anchor=(1.05, 1), loc="upper left")
plt.show()
```



6.1 Set 3 (D Tasks) Analyse the learned representations

(weight ~ 10%)

In this task, you will explore the visualization of embeddings at different layers of your trained neural network and analyse how they evolve using **Uniform Manifold Approximation and Projection** (UMAP). Below are detailed steps you can follow.

1. Select a subset of your training data containing both classes.
2. Extract the embeddings from each layer of the neural network model for the dataset.
3. Apply UMAP to visualise the embeddings from each layer in a 2-dimensional space, highlighting different classes with distinct colours or markers. Include appropriate labels and legends in your plots.
4. Analyse and discuss the evolution of the embeddings across layers. Answer the following questions in your analysis:

- Do the embeddings show a clear separation between classes at any specific layer?
- How do the separation and clustering of classes change as you move across layers?
- Visualise the embeddings at the beginning, midway during training, and at the end of training. Comment on what you observe.
- Are there any notable changes in the distribution or structure of the embeddings?
- Are there any layers where the embeddings become less discriminative or more entangled?

5. Summarize your findings and provide insights into the behaviour of the neural network"s representations at different layers. Discuss the implications of the observed changes in the embeddings for the network"s ability to capture class-specific information and make predictions.

Answer:

4. Analysis:

- Do the embeddings show a clear separation between classes at any specific layer?
  - In Output Layer (Layer 3): Embeddings shows the most distinct clusters for different classes. The embeddings are designed to be highly discriminative, so products like bag, sandal and trouser are well-separated in this layer. Although products with similar appearances might still overlap, the clustering is better than layer 2 and far more bettern than layer 1.
- How do the separation and clustering of classes change as you move across layers?
  - Layer 1: The embeddings in this layer captures low-level features such as edges and textures. The separation between classes is less pronounced although classes with similarities are already concentrated into a big cluster, and clusters still overlap significantly. In this layer, we can recoginize bag and trouser are separated well to some extent.
  - Layer 2: The embeddings capture more abstract and high-level features. Clustering improves, and separation between classes becomes more apparent, though some overlap still occur for similar products. In this layer, we can recognize sandal has been separated also.
  - Output Layer: The embeddings are designed to provide clear separation between classes. This layer shows the best clustering and separation such as bag, trouser, sandal. However, because of similarities in shapes and textures, the network still finds it challenging to distinguish some similar classes such as T-shirt, shirt, pullover, etc. In this layer, although there are still overlap regions in between, but for most of each class, we can already see their improved own cluster.
- Are there any notable changes in the distribution or structure of the embeddings?
  - Layer 1: The embeddings are more spread out and less structured. The model captures basic features without much discrimination between classes.
  - Layer 2: The distribution becomes more organized as the model learns higher-level features. Clusters start to form more distinctly, reflecting improved class separation.
  - Layer 3: Embeddings become tightly clustered around their respective classes, showing better separated groups, reducing overlapping area.
- Are there any layers where the embeddings become less discriminative or more entangled?
  - No, we can see clearly that the classification is better over layers.

5. Summary:

- The visualization shows a progressive improvement in the model's ability to learn/capture different classes. Layer 1, although it has done well in cluster products based on their textures,

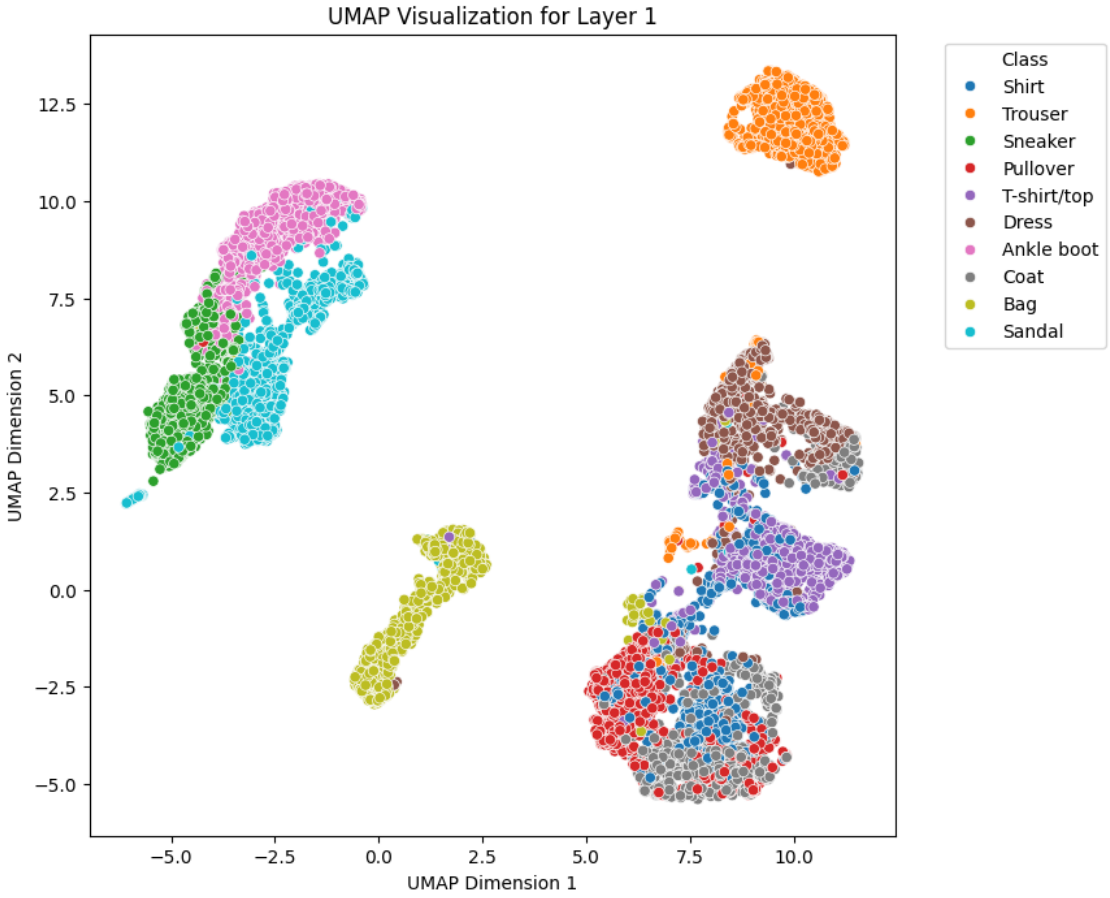
it still exhibits a lot of overlapping embeddings between similar items. Layer 2 shows better clustering and separation of categories, indicating that the model is learning more abstract features that enhance class discrimination. Layer 3 shows that embeddings are well-defined with clear clusters. However, some overlap among similar classes persists, but it is reduced compared to the previous layers.

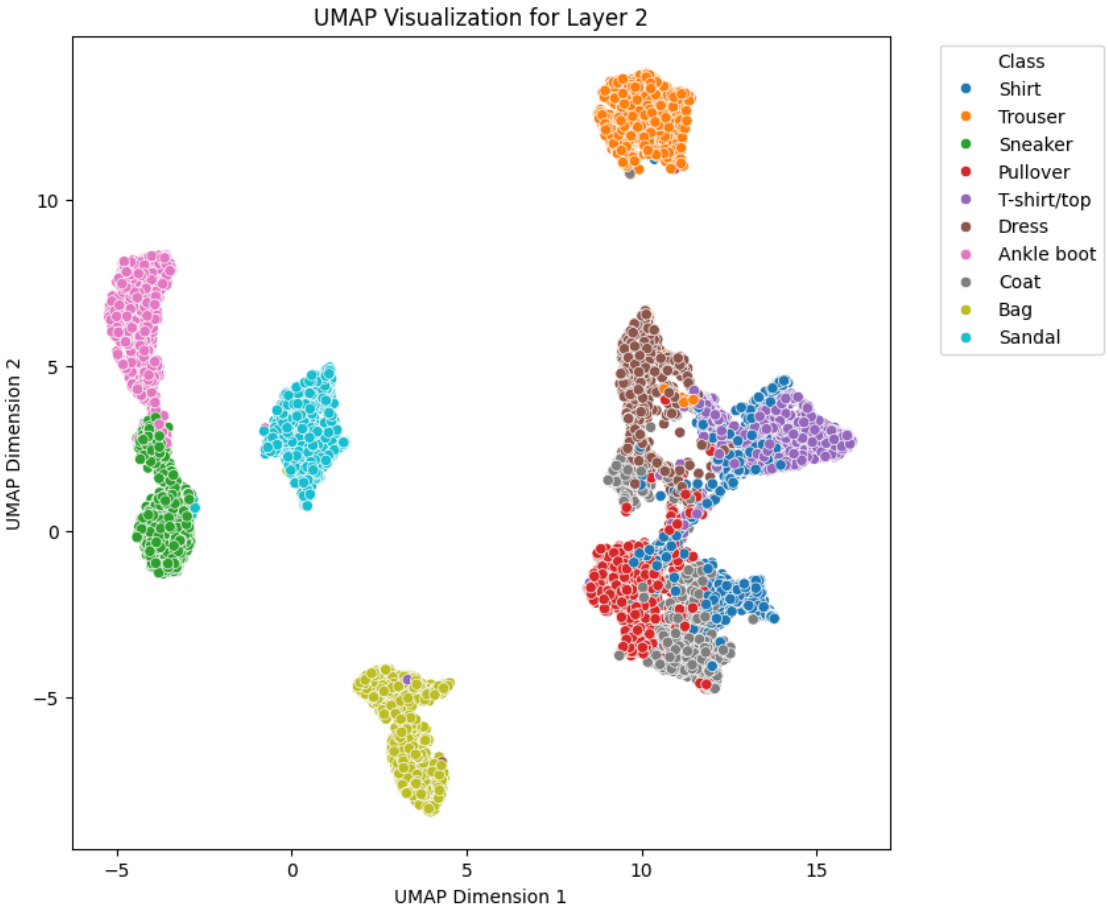
- This evolution underscores the network's growing proficiency in feature abstraction and highlights the potential for further model optimization to address challenges in distinguishing closely related classes.

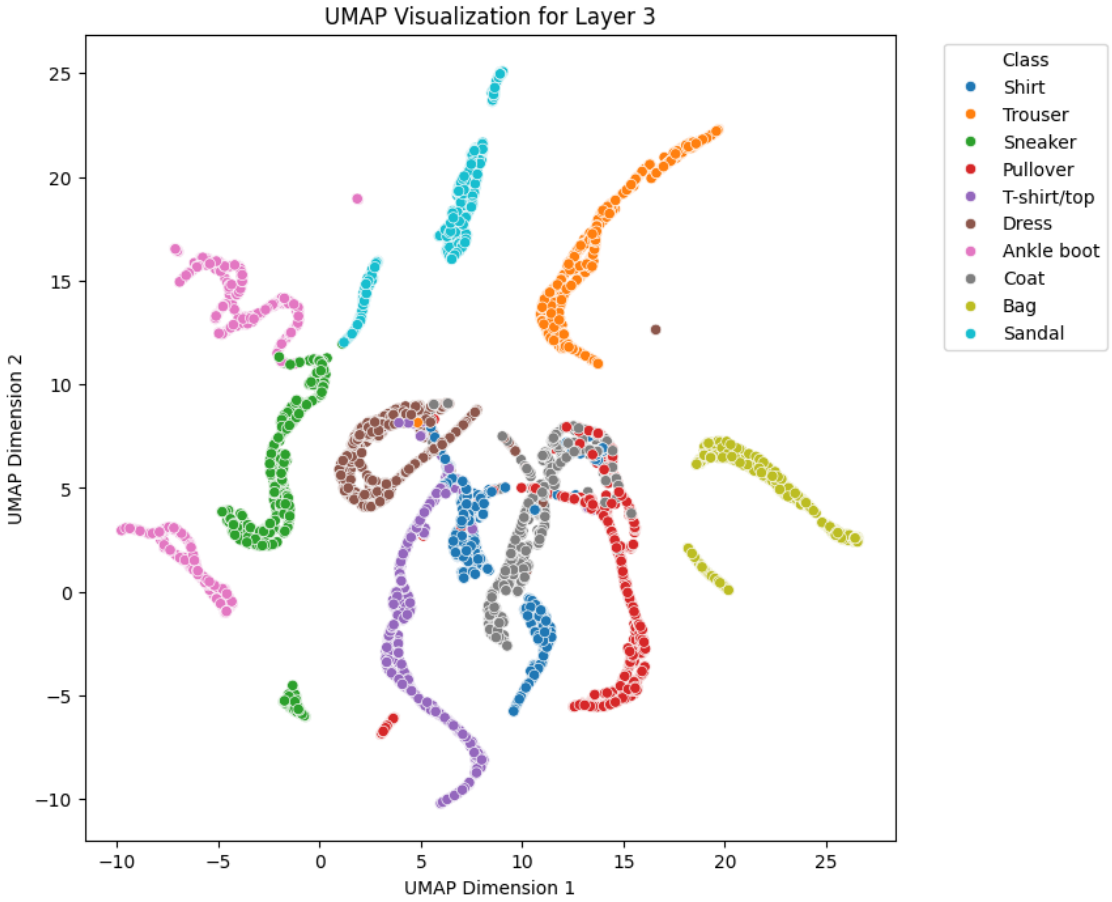
[32]:
```
# Select a subset of training data (20%) containing all classes.
_, subset_images, _, subset_labels = train_test_split(train_images,␣
↪train_labels, test_size=0.2, stratify=train_labels)
```

[33]:
```
# Extract the outputs of each layer in the model
output_layers = [layer.output for layer in model.layers]
embedding_model = Model(inputs=model.input, outputs=output_layers)

# Get the embeddings for the subset of data
embeddings = embedding_model.predict(subset_images, verbose=0)
```

[34]:
```
for i, embedding in enumerate(embeddings):
  # Apply UMAP to reduce dimensionality
  transformed_embedding = umap.UMAP().fit_transform(embedding)

  # Get class names
  class_labels = [class_names[label] for label in np.argmax(subset_labels,␣
 ↪axis=1)]

  # Plot UMAP embeddings
  plt.figure(figsize=(8, 8))
  sns.scatterplot(x=transformed_embedding[:, 0], y=transformed_embedding[:, 1],␣
 ↪hue=class_labels, palette="tab10")
  plt.title(f"UMAP Visualization for Layer {i+1}")
  plt.xlabel("UMAP Dimension 1")
  plt.ylabel("UMAP Dimension 2")
  plt.legend(title="Class", bbox_to_anchor=(1.05, 1), loc="upper left")
  plt.show()
```


UMAP Visualization for Layer 1
UMAP Dimension 2
UMAP Dimension 1
12.5
10.0
7.5
5.0
2.5
0.0
−2.5
−5.0
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0
Class
Shirt
Trouser
Sneaker
Pullover
T-shirt/top
Dress
Ankle boot
Coat
Bag
Sandal


UMAP Visualization for Layer 2
UMAP Dimension 2
UMAP Dimension 1
10
5
0
−5
−5
0
5
10
15
Class
Shirt
Trouser
Sneaker
Pullover
T-shirt/top
Dress
Ankle boot
Coat
Bag
Sandal



6.2 Set 4 (HD Tasks) Investigating Glorot Initialisation

(weight ~20%)

Read the paper "Understanding the difficulty of training deep feedforward neural networks". (https://proceedings.mlr.press/v9/glorot10a/glorot10a.pdf).

6.2.1 Task 4.1 Conceptual Understanding

1. What problem does the paper address?
2. What is the authors" hypothesis and how do they develop and validate it? Feel free to explain your understanding of the paper with your own illustrations/block diagrams etc.,
3. List and briefly summarise some recent works which have built on the idea in this paper. Cite all relevant references.

Answer:

1. What problem does the paper address?

- The paper explores why deep neural networks often perform poorly when trained with randomly initialized weights.

- The paper also examines how does vanishing/exploding gradient problem occur during backpropagation, hindering effective learning.
- The paper highlights how proper initialization, like normalized initialization (Glorot initialization), is crucial to stabilize activation, backpropagated gradients variance and prevent gradient-related issues.
- The paper investigates how different activation functions such as Sigmoid, Tanh, Soft Sign affect the saturation of units and the training process.
- The paper also emphasizes the importance of using appropriate cost functions, in classification tasks, to enhance learning efficiency.

2. What is the authors" hypothesis and how do they develop and validate it? Feel free to explain your understanding of the paper with your own illustrations/block diagrams etc.,

- The authors hypothesize that deep neural networks struggle with training due to the vanishing or exploding gradients problem, which is caused by poor weight initialization. They propose that proper weight initialization and activation function is crucial for solving the problem. To validate this, they introduce the normalized initialization method, designed to keep the variance of activations and gradients constant across layers. They also monitor the values of the Jacobian matrix across layers. The hypothesis has been validated and is shown in below figure 1, 2
- Figure 1

- Figure 2

- Through experiments with synthetic and real-world datasets, the authors demonstrate that normalized initialization significantly improves training stability and performance, especially when used with activation functions like Tanh or Soft Sign, which mitigate gradient issues better than functions like Sigmoid because of its symmetry around 0 as table 1 below.
- Table 1

1. List and briefly summarise some recent works which have built on the idea in this paper. Cite all relevant references.

- Paper "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification"
  - **Summary:** He initialization was developed to address the shortcomings of the normalized initialization method. Although Glorot initialization, based on the assumption of linear activation functions, maintains variance across layers for activation functions like sigmoid and tanh, it is less suited for ReLU activations. Due to their non-negative nature, ReLU functions can be benefit more from He initialization.
  - **Reference:** He, K., Zhang, X., Ren, S., & Sun, J. (2015). Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. 2015 IEEE International Conference on Computer Vision (ICCV). DOI: 10.1109/ICCV.2015.123.

- Paper "Feedforward Neural Networks Initialization Based on Discriminant Learning"
  - **Summary:** This paper introduces a novel initialization scheme based on discriminant learning. This research extends the concept of normalized initialization by incorporating discriminant learning principles. Its approach adjusts the initialization process to better account for the specific characteristics of the target classification task. This refinement aims to improve convergence speed and overall network performance by considering some of the limitations of the existing method caused by assumptions on the data and proposing ways to remedy them.
  - **Reference:** Kateryna, C., Alexandros, I., Moncef, F. (2022). Feedforward Neural Networks Initialization Based on Discriminant Learning. Neural Networks, Volume 146. DOI:10.1016/j.neunet.2021.11.020.
- Paper "All You Need is a Good Init"
  - **Summary:** The paper emphasizes the crucial role of proper initialization in achieving efficient and stable training. It reviews several initialization techniques, including Glorot initialization, and explores their effects on network convergence and learning dynamics and their limitations, assumptions. The authors then introduce new initialization strategies that improve upon existing methods by enhancing training stability and convergence speed.
  - **Reference:** Dmytro, M., Jiri, M.. (2016). All You Need is a Good Init. 2016 Conference at ICLR. DOI: 10.48550/arXiv.1511.06422.

6.2.2 Task 4.2 Implementation

1. Implement Glorot Initialisation in the Fashion-MNIST classification problem.
2. Compare the performance and convergence rates to the original implementation without Glorot Initialisation. Do you observe any differences? Why/why not?

Answer:

- In TensorFlow, the default kernel_initializer is set to Glorot Uniform. Therefore, I will construct the network using Glorot initialization (the result is expected to be near to the pass task, there can be some slight different because the weight initialization still follows uniform distribution) and compare it to a network using traditional random initialization (the result may have slower convergence speed and lower accuracy). In the traditional approach, kernel_initializer follows a uniform distribution, generating random weights within the range [-1/sqrt(n), 1/sqrt(n)], where n is the size of the previous layer (or the input size of the current layer).
- However, as mentioned, because of similarities between products and I just use 3 layers in total (2 hidden layers, 1 output layer), the different between glorot initialization and standard initialization may not be significant.

Factor	Glorot Initialization	Standard Initialization
Test Loss	0.384	0.379
Test Accuracy	0.894	0.890
Training Time	127 s	143 s

- Overall, Glorot Initialization shows a slightly higher test loss (0.384) compared to Standard Initialization (0.379), indicating Standard Initialization may have better generalization .

- The test accuracy with Glorot Initialization is higher at 89.4%, compared to 89% with Standard Initialization.
- Additionally, the training time is reduced with Glorot Initialization, taking 127 seconds versus 143 seconds with Standard Initialization, suggesting that Glorot Initialization can lead to more efficient training.
- Insight:
  - With only two hidden layers, each with 512 neurons, the network is simple. The impact of Glorot Initialization might be less pronounced. It will be more noticeable in deeper networks where maintaining gradient stability is crucial.
  - ReLU activation functions are less sensitive to the choice of weight initialization compared to activation functions like sigmoid or tanh.
  - Adam optimizer adjusts learning rates during training, which can mitigate the vanishing/exploding gradient problem.

In addition to short answers to the above questions, submit a short (less than 5 minutes) video presentation for your analysis and main conclusions.

[35]:
```
# Configure layers' number of neurons, activation functions, glorot_uniform␣
 ↪initialization
model_glorot = Sequential([layers.Dense(512, activation="relu",␣
 ↪input_shape=(28*28,), kernel_initializer="glorot_uniform"),
                           layers.Dense(512, activation="relu",␣
 ↪kernel_initializer="glorot_uniform"),
                           layers.Dense(10, activation="softmax",␣
 ↪kernel_initializer="glorot_uniform")])

# Configure model's optimizer, loss function and metrics
model_glorot.compile(optimizer="Adam",
                     loss="categorical_crossentropy",
                     metrics=["accuracy"])

# Calculate training time
start = time.time()

# Fit train/validate dataset to configured model
history = model_glorot.fit(train_images,
                           train_labels,
                           epochs=20,
                           batch_size=128,
                           validation_data=(val_images, val_labels),
                           verbose=0)

# Calculate training time
print("Training time:", time.time() - start)

# Evaluate model's performance
```

```
test_loss, test_accuracy = model_glorot.evaluate(test_images, test_labels,␣
 ↪verbose=0)

print("Test Loss:", test_loss)
print("Test Accuracy:", test_accuracy)
```

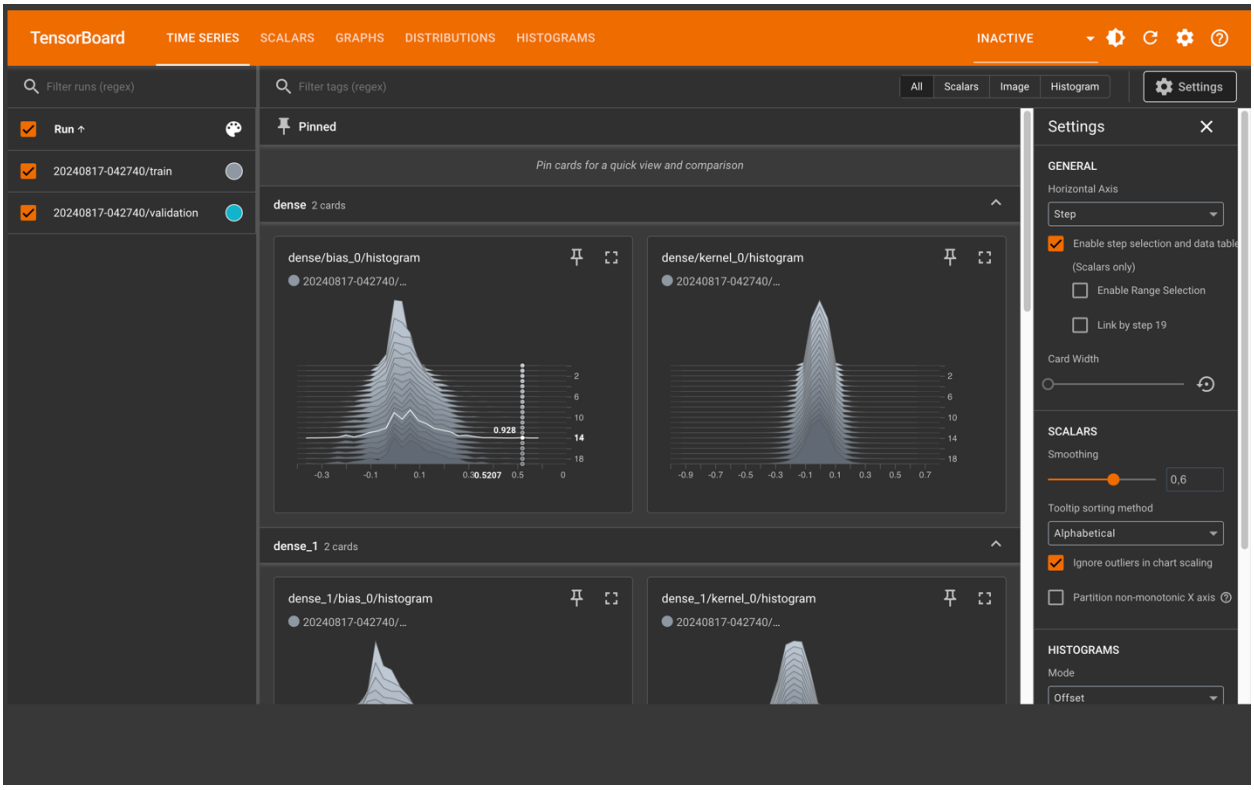
```
Training time: 126.62912106513977
Test Loss: 0.3840211033821106
Test Accuracy: 0.8937000036239624
```

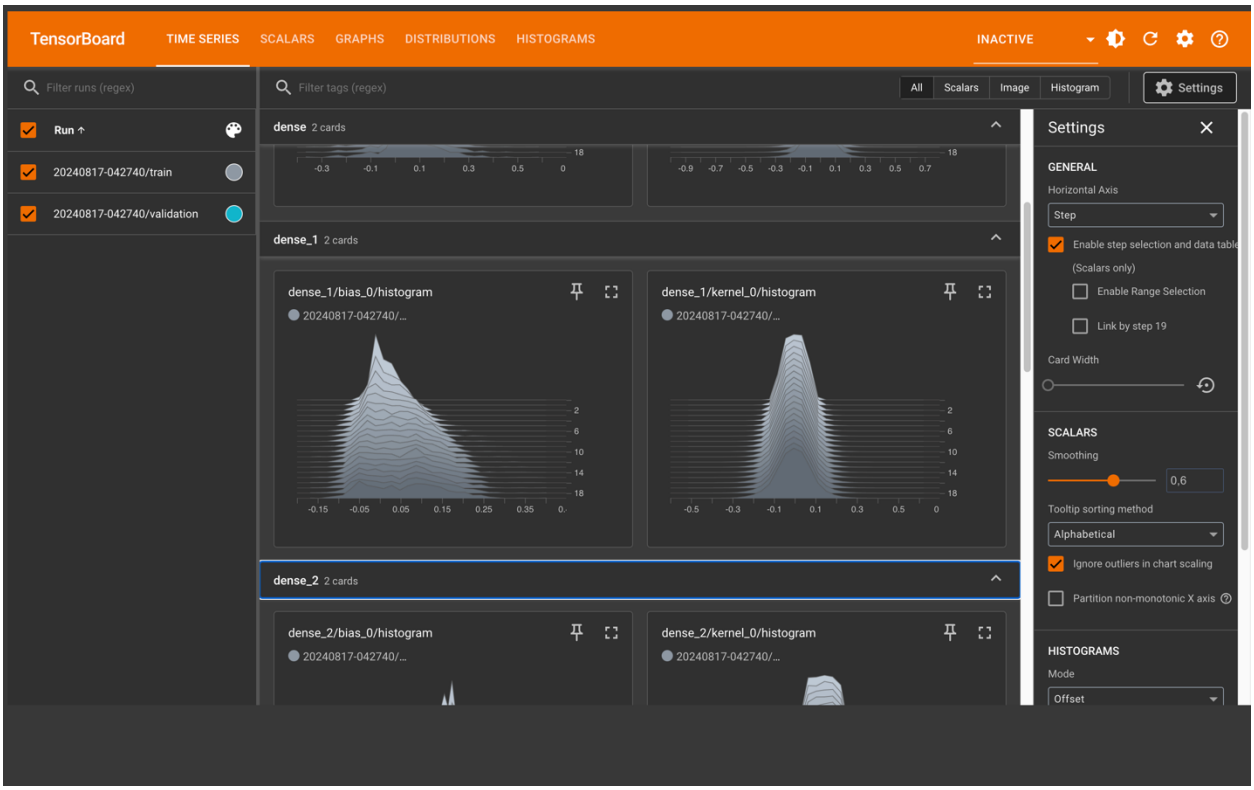
[38]:
```
# Configure layers' number of neurons, activation functions, standard␣
 ↪initialization
model_standard = Sequential([layers.Dense(512, activation="relu",␣
 ↪input_shape=(28*28,), kernel_initializer=RandomUniform(-1/(784**0.5), 1/
 ↪(784**0.5))),
                             layers.Dense(512, activation="relu",␣
 ↪kernel_initializer=RandomUniform(-1/(512**0.5), 1/(512**0.5))),
                             layers.Dense(10, activation="softmax",␣
 ↪kernel_initializer=RandomUniform(-1/(512**0.5), 1/(512**0.5)))])

# Configure model's optimizer, loss function and metrics
model_standard.compile(optimizer="Adam",
                       loss="categorical_crossentropy",
                       metrics=["accuracy"])

# Calculate training time
start = time.time()

# Fit train/validate dataset to configured model
history = model_standard.fit(train_images,
                             train_labels,
                             epochs=20,
                             batch_size=128,
                             validation_data=(val_images, val_labels),
                             verbose=0)

# Calculate training time
print("Training time:", time.time() - start)

# Evaluate model's performance
test_loss, test_accuracy = model_standard.evaluate(test_images, test_labels,␣
 ↪verbose=0)

print("Test Loss:", test_loss)
print("Test Accuracy:", test_accuracy)
```

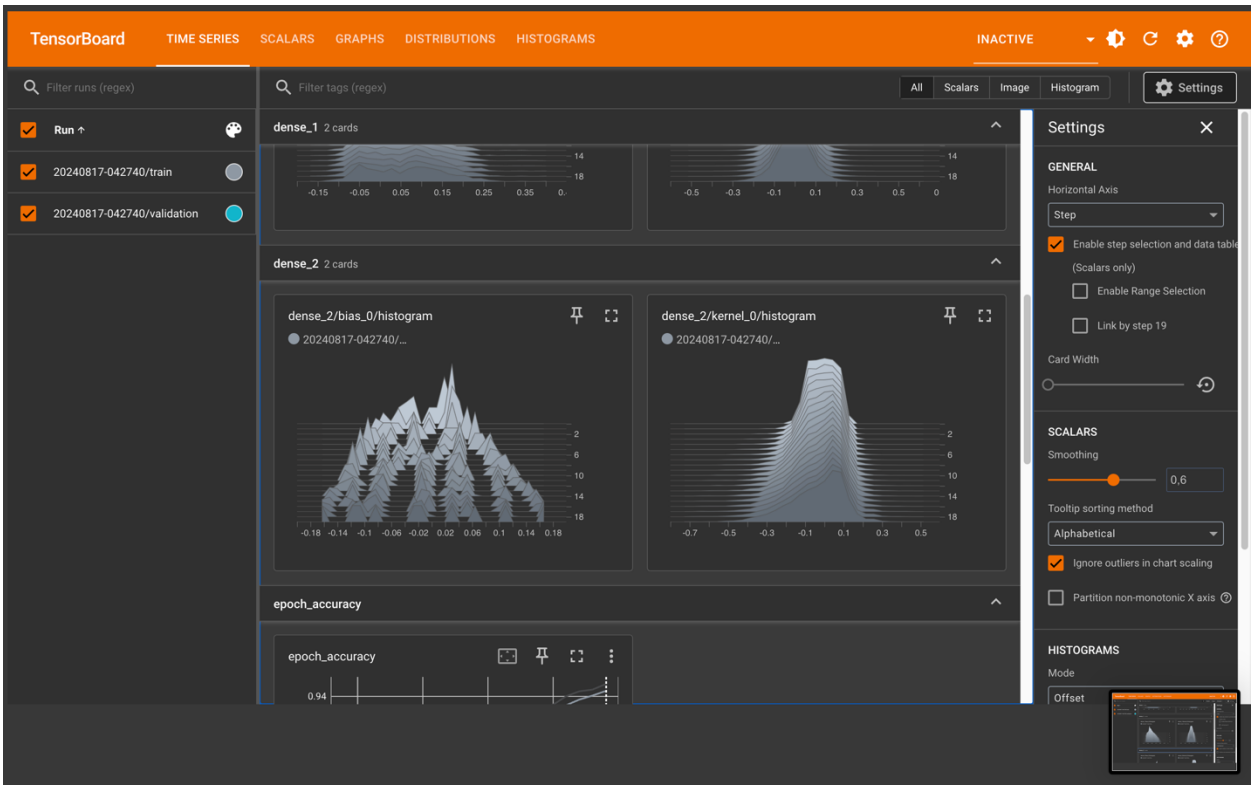
```
Training time: 142.95604920387268
Test Loss: 0.37927305698394775
```

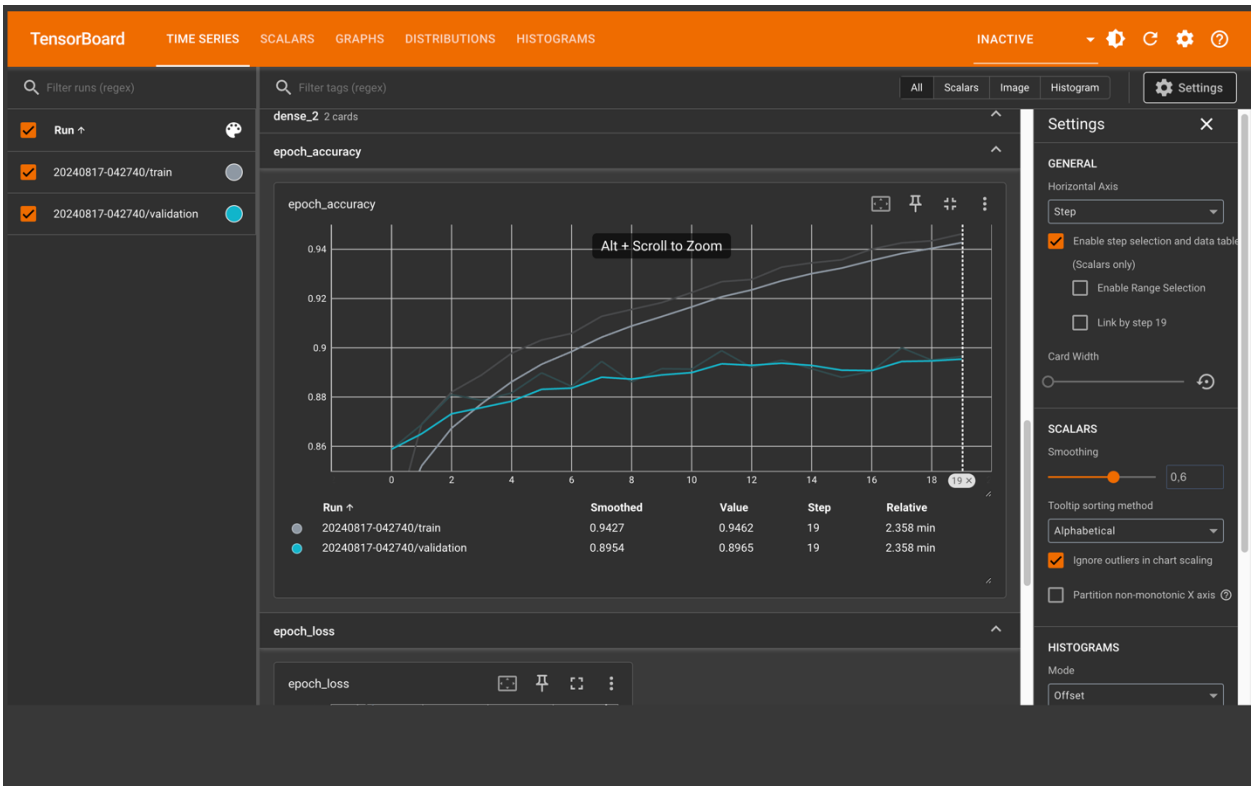
```
Test Accuracy: 0.8896999955177307
```

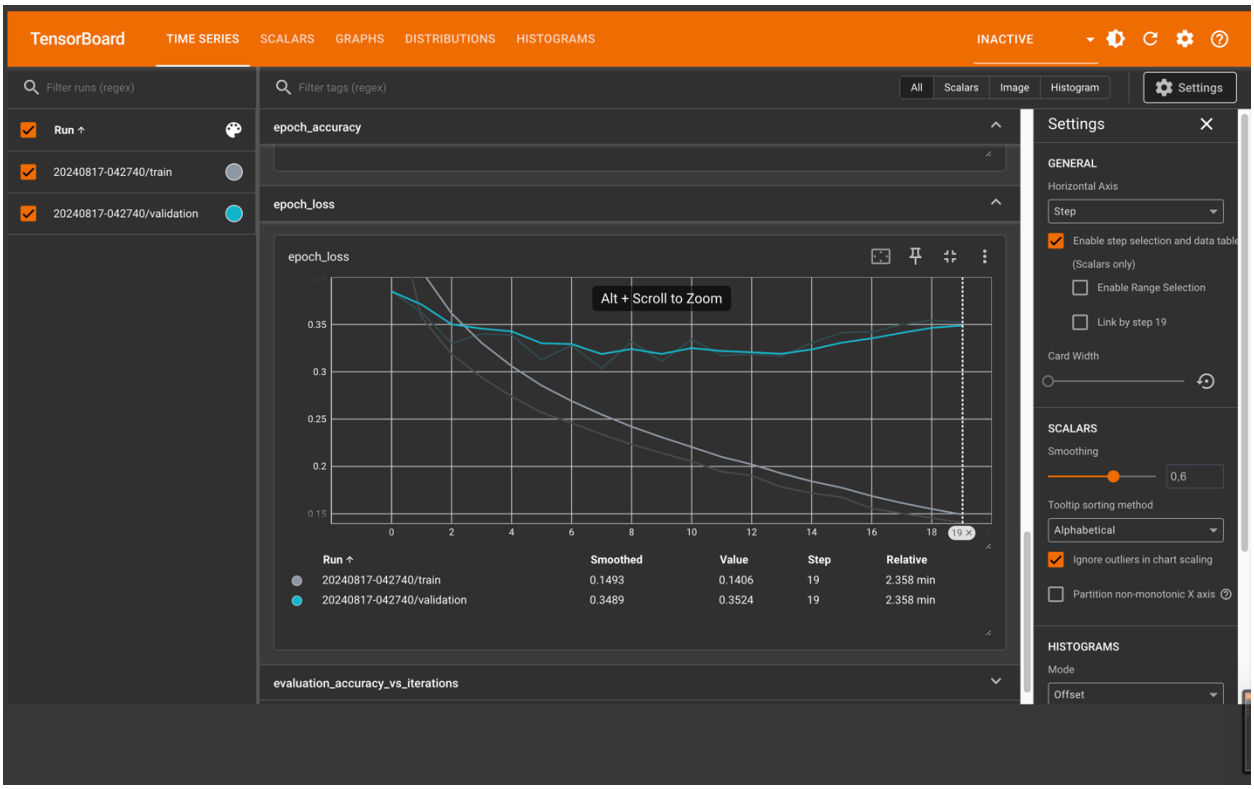
TensorBoard Result

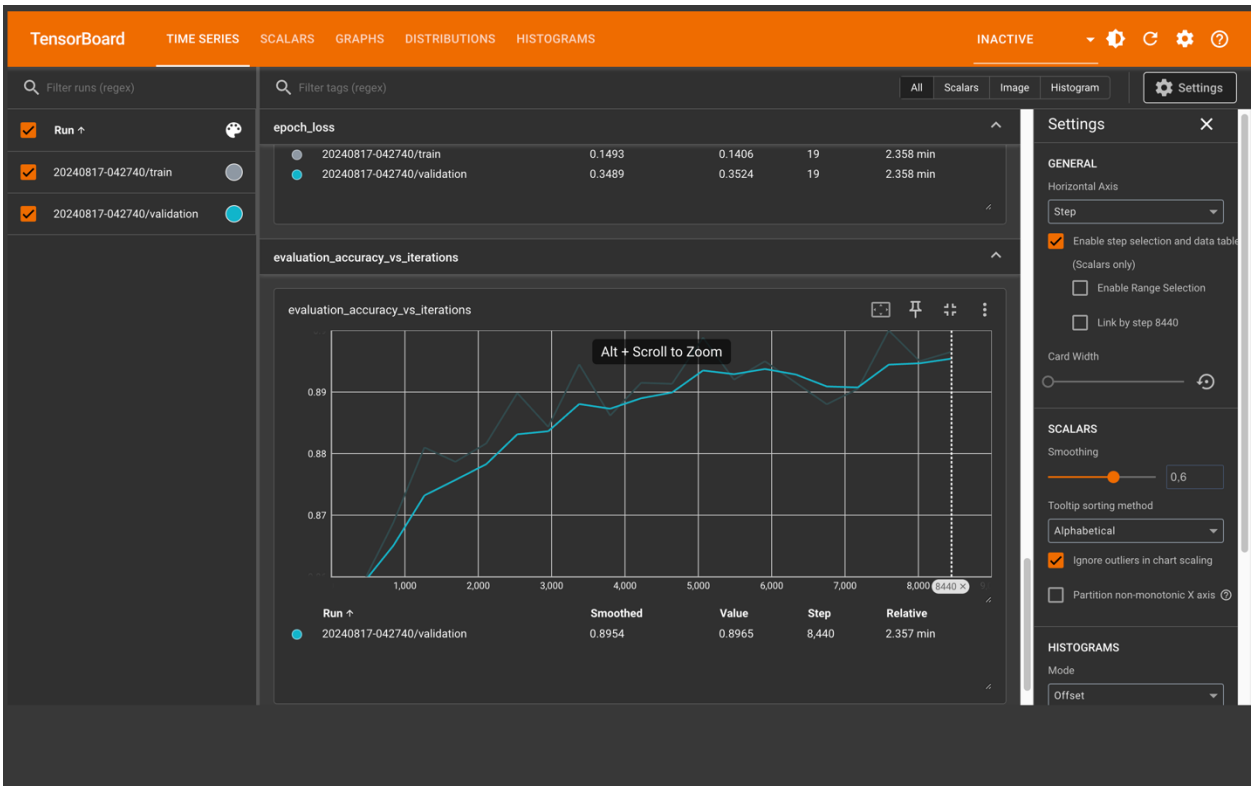


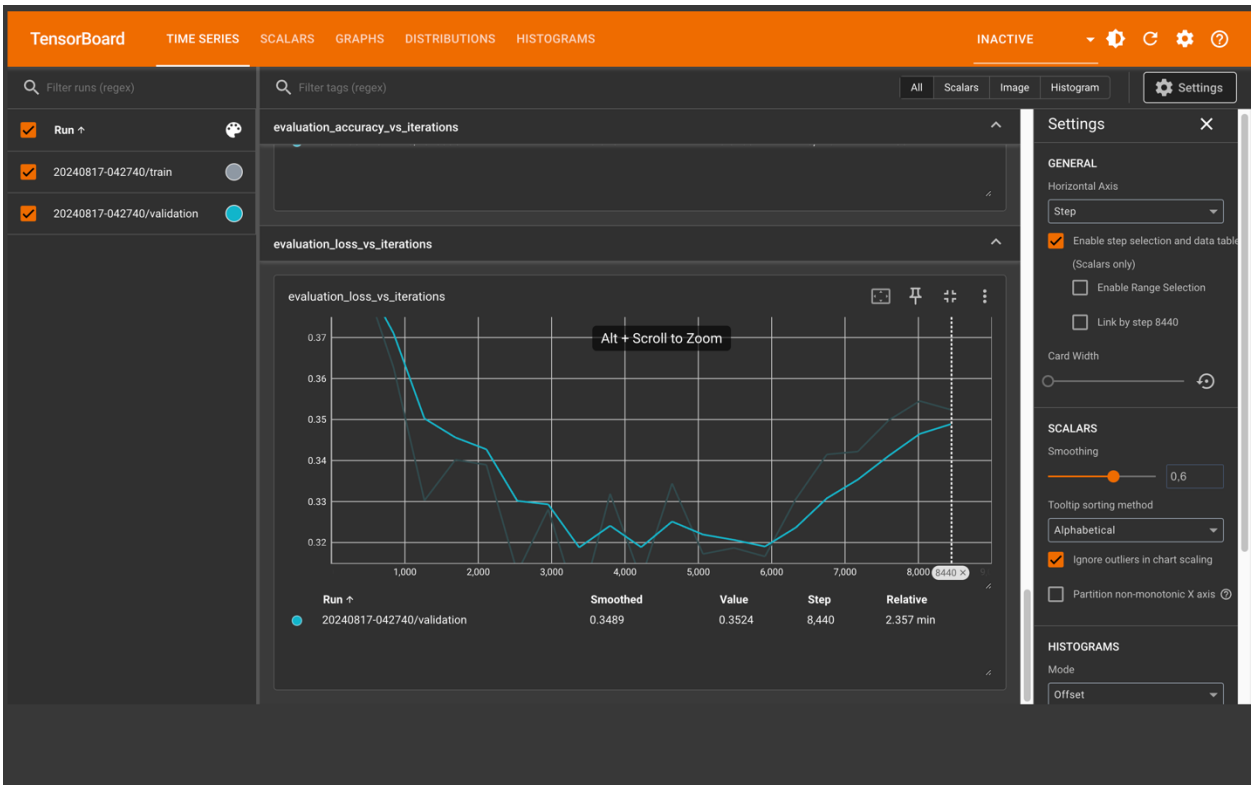


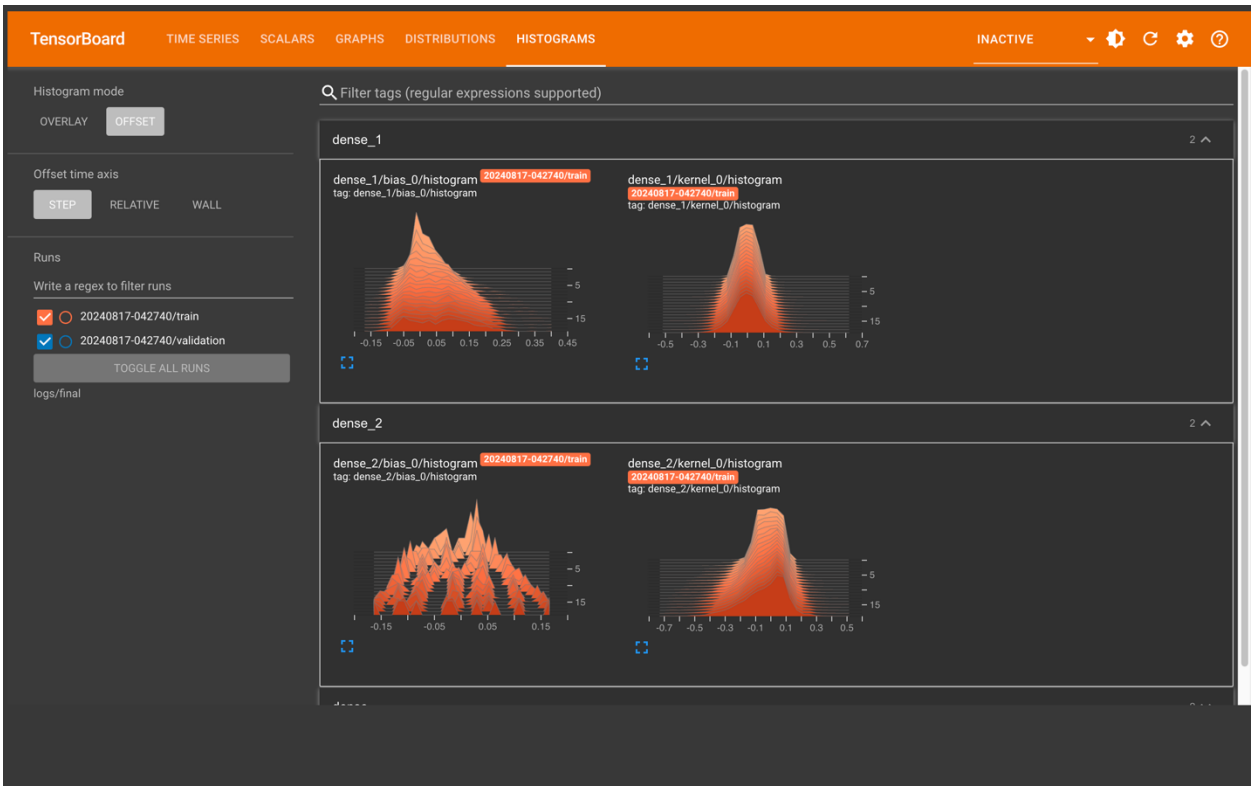

TensorBoard
TIME SERIES
SCALARS
GRAPHS
DISTRIBUTIONS
HISTOGRAMS
INACTIVE
Filter runs (regex)
Filter tags (regex)
All
Scalars
Image
Histogram
Settings
Run
20240817-042740/train
20240817-042740/validation
dense_1 2 cards
dense_2 2 cards
dense_2/bias_0/histogram
dense_2/kernel_0/histogram
epoch_accuracy
GENERAL
Horizontal Axis
Step
Enable step selection and data table
(Scalars only)
Enable Range Selection
Link by step 19
Card Width
SCALARS
Smoothing
0,6
Tooltip sorting method
Alphabetical
Ignore outliers in chart scaling
Partition non-monotonic X axis
HISTOGRAMS
Mode
Offset

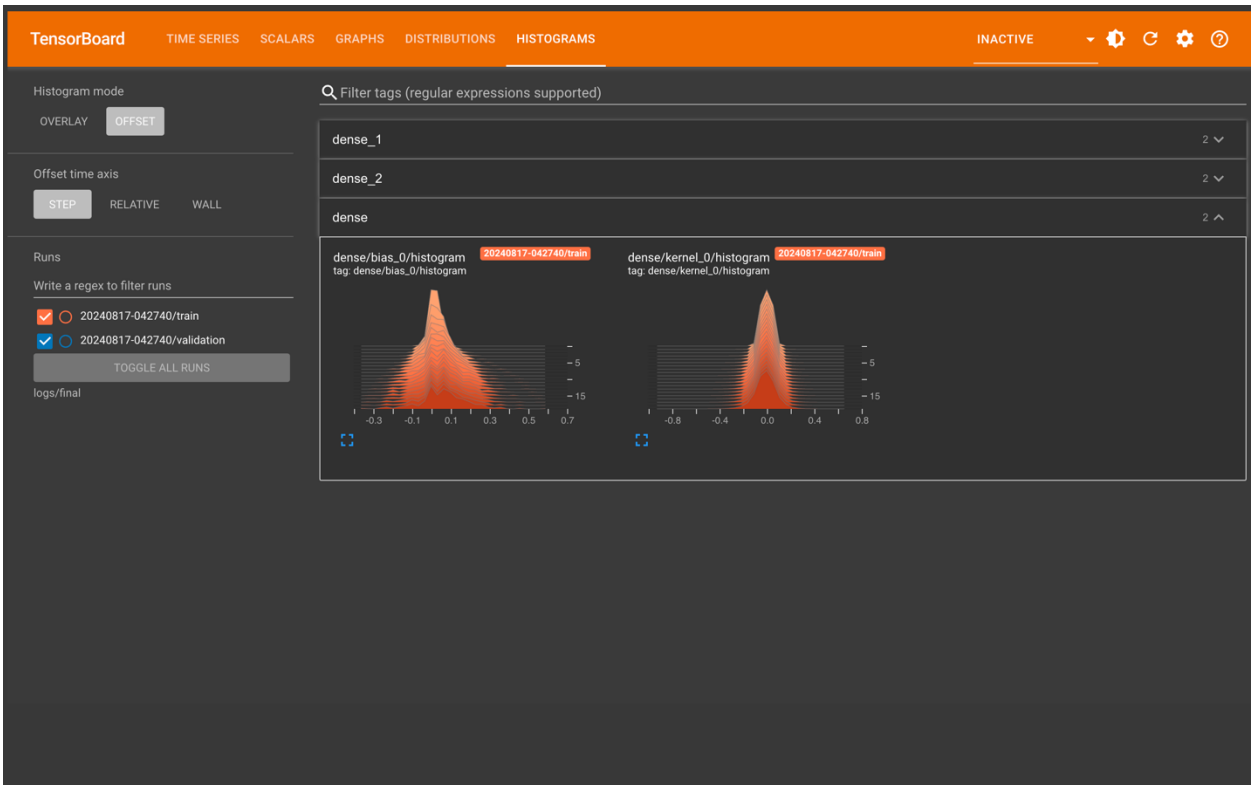

TensorBoard
TIME SERIES
SCALARS
GRAPHS
DISTRIBUTIONS
HISTOGRAMS
INACTIVE
Filter runs (regex)
Filter tags (regex)
All
Scalars
Image
Histogram
Settings
Run
20240817-042740/train
20240817-042740/validation
dense_2 2 cards
epoch_accuracy
Alt + Scroll to Zoom
Run
Smoothed
Value
Step
Relative
20240817-042740/train 0.9427 0.9462 19 2.358 min
20240817-042740/validation 0.8954 0.8965 19 2.358 min
epoch_loss
GENERAL
Horizontal Axis
Step
Enable step selection and data table
(Scalars only)
Enable Range Selection
Link by step 19
Card Width
SCALARS
Smoothing
0,6
Tooltip sorting method
Alphabetical
Ignore outliers in chart scaling
Partition non-monotonic X axis
HISTOGRAMS
Mode
Offset

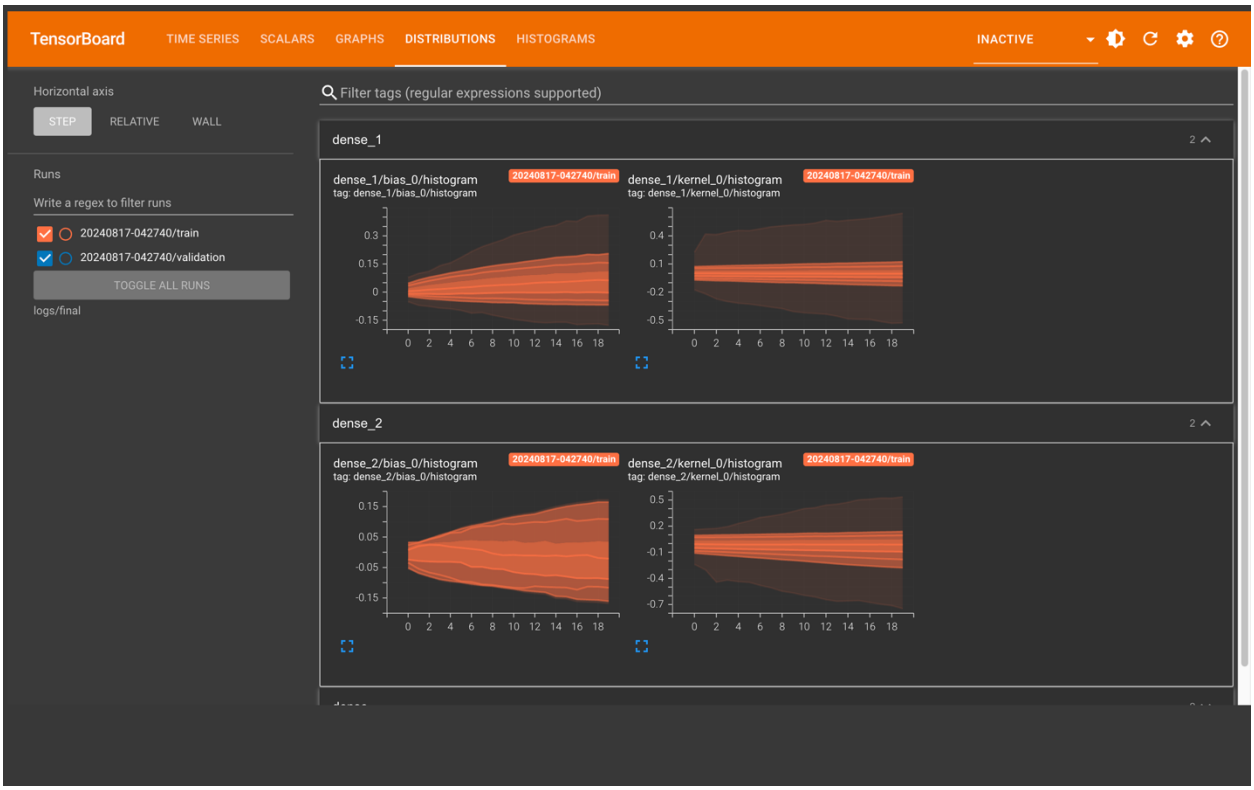

TensorBoard
TIME SERIES
SCALARS
GRAPHS
DISTRIBUTIONS
HISTOGRAMS
INACTIVE
Filter runs (regex)
Filter tags (regex)
All
Scalars
Image
Histogram
Settings
Run
20240817-042740/train
20240817-042740/validation
epoch_accuracy
epoch_loss
Alt + Scroll to Zoom
0.35
0.3
0.25
0.2
0.15
0
2
4
6
8
10
12
14
16
18
19
Run
Smoothed
Value
Step
Relative
20240817-042740/train 0.1493 0.1406 19 2.358 min
20240817-042740/validation 0.3489 0.3524 19 2.358 min
evaluation_accuracy_vs_iterations
Settings
GENERAL
Horizontal Axis
Step
Enable step selection and data table
(Scalars only)
Enable Range Selection
Link by step 19
Card Width
SCALARS
Smoothing
0,6
Tooltip sorting method
Alphabetical
Ignore outliers in chart scaling
Partition non-monotonic X axis
HISTOGRAMS
Mode
Offset


TensorBoard
TIME SERIES
SCALARS
GRAPHS
DISTRIBUTIONS
HISTOGRAMS
INACTIVE
Filter runs (regex)
Filter tags (regex)
All
Scalars
Image
Histogram
Settings
Run
20240817-042740/train
20240817-042740/validation
epoch_loss
20240817-042740/train 0.1493 0.1406 19 2.358 min
20240817-042740/validation 0.3489 0.3524 19 2.358 min
evaluation_accuracy_vs_iterations
Alt + Scroll to Zoom
0.89
0.88
0.87
1,000
2,000
3,000
4,000
5,000
6,000
7,000
8,000
8440
Run
Smoothed
Value
Step
Relative
20240817-042740/validation 0.8954 0.8965 8,440 2.357 min
Settings
GENERAL
Horizontal Axis
Step
Enable step selection and data table
(Scalars only)
Enable Range Selection
Link by step 8440
Card Width
SCALARS
Smoothing
0,6
Tooltip sorting method
Alphabetical
Ignore outliers in chart scaling
Partition non-monotonic X axis
HISTOGRAMS
Mode
Offset


TensorBoard
TIME SERIES
SCALARS
GRAPHS
DISTRIBUTIONS
HISTOGRAMS
INACTIVE
Filter runs (regex)
Filter tags (regex)
All
Scalars
Image
Histogram
Settings
Run
20240817-042740/train
20240817-042740/validation
evaluation_accuracy_vs_iterations
evaluation_loss_vs_iterations
Alt + Scroll to Zoom
0.37
0.36
0.35
0.34
0.33
0.32
1,000
2,000
3,000
4,000
5,000
6,000
7,000
8,000
8440
Run
Smoothed
Value
Step
Relative
20240817-042740/validation
0.3489
0.3524
8,440
2.357 min
Settings
GENERAL
Horizontal Axis
Step
Enable step selection and data table
(Scalars only)
Enable Range Selection
Link by step 8440
Card Width
SCALARS
Smoothing
0,6
Tooltip sorting method
Alphabetical
Ignore outliers in chart scaling
Partition non-monotonic X axis
HISTOGRAMS
Mode
Offset


TensorBoard
TIME SERIES
SCALARS
GRAPHS
DISTRIBUTIONS
HISTOGRAMS
INACTIVE
Histogram mode
OVERLAY
OFFSET
Offset time axis
STEP
RELATIVE
WALL
Runs
Write a regex to filter runs
20240817-042740/train
20240817-042740/validation
TOGGLE ALL RUNS
logs/final
Filter tags (regular expressions supported)
dense_1
dense_1/bias_0/histogram
20240817-042740/train
tag: dense_1/bias_0/histogram
dense_1/kernel_0/histogram
20240817-042740/train
tag: dense_1/kernel_0/histogram
dense_2
dense_2/bias_0/histogram
20240817-042740/train
tag: dense_2/bias_0/histogram
dense_2/kernel_0/histogram
20240817-042740/train
tag: dense_2/kernel_0/histogram


TensorBoard
TIME SERIES
SCALARS
GRAPHS
DISTRIBUTIONS
HISTOGRAMS
INACTIVE
Histogram mode
OVERLAY
OFFSET
Offset time axis
STEP
RELATIVE
WALL
Runs
Write a regex to filter runs
20240817-042740/train
20240817-042740/validation
TOGGLE ALL RUNS
logs/final
Filter tags (regular expressions supported)
dense_1
dense_2
dense
dense/bias_0/histogram
tag: dense/bias_0/histogram
20240817-042740/train
dense/kernel_0/histogram
tag: dense/kernel_0/histogram
20240817-042740/train


TensorBoard
TIME SERIES
SCALARS
GRAPHS
DISTRIBUTIONS
HISTOGRAMS
INACTIVE
Horizontal axis
STEP
RELATIVE
WALL
Runs
Write a regex to filter runs
20240817-042740/train
20240817-042740/validation
TOGGLE ALL RUNS
logs/final
Filter tags (regular expressions supported)
dense_1
dense_1/bias_0/histogram
tag: dense_1/bias_0/histogram
20240817-042740/train
dense_1/kernel_0/histogram
tag: dense_1/kernel_0/histogram
20240817-042740/train
dense_2
dense_2/bias_0/histogram
tag: dense_2/bias_0/histogram
20240817-042740/train
dense_2/kernel_0/histogram
tag: dense_2/kernel_0/histogram
20240817-042740/train

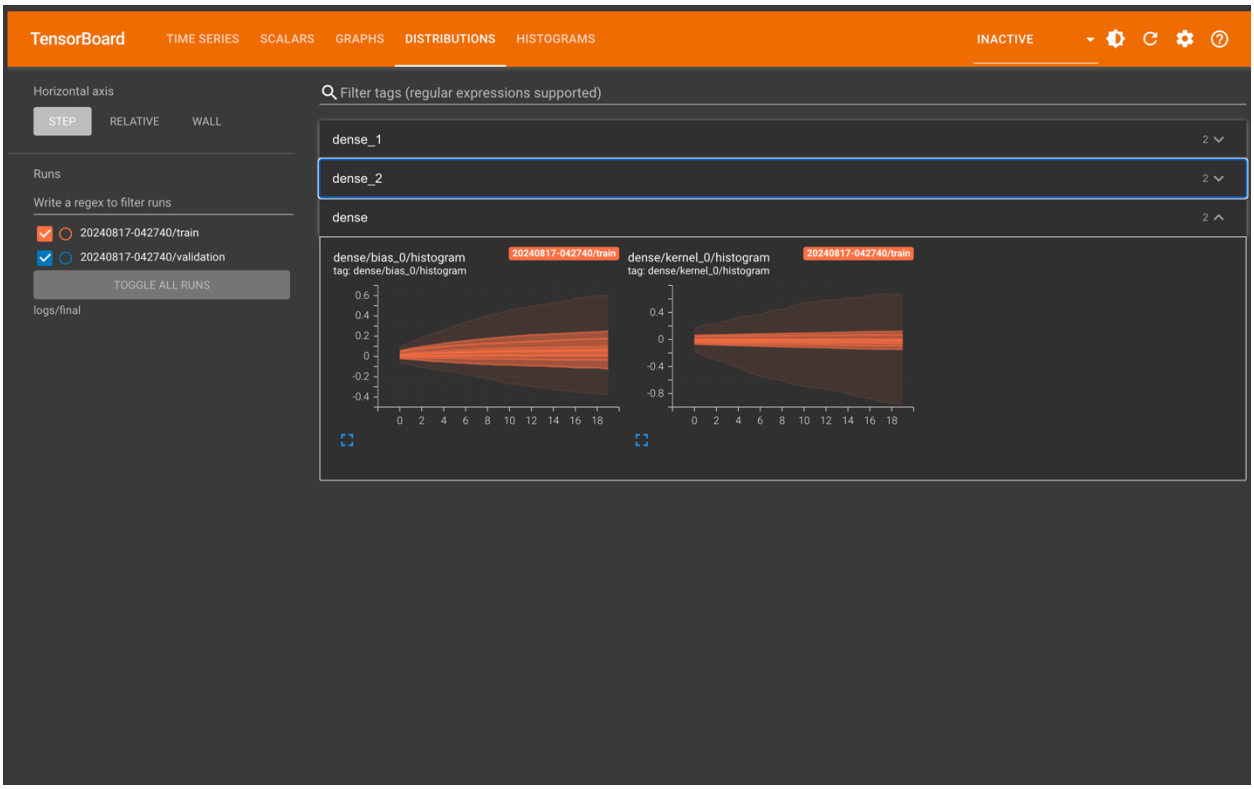

TensorBoard
TIME SERIES
SCALARS
GRAPHS
DISTRIBUTIONS
HISTOGRAMS
INACTIVE
Horizontal axis
STEP
RELATIVE
WALL
Runs
Write a regex to filter runs
20240817-042740/train
20240817-042740/validation
TOGGLE ALL RUNS
logs/final
Filter tags (regular expressions supported)
dense_1
2
dense_2
2
dense
2
dense/bias_0/histogram
tag: dense/bias_0/histogram
20240817-042740/train
dense/kernel_0/histogram
tag: dense/kernel_0/histogram
20240817-042740/train

Figure 1

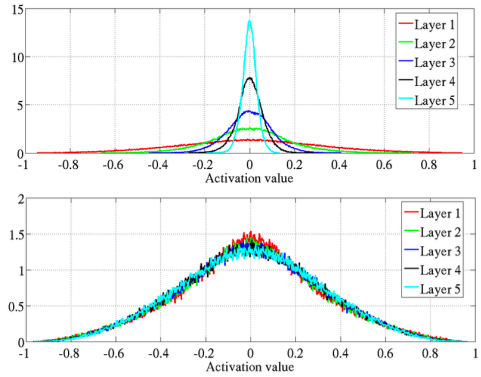


Figure 6: *Activation values normalized histograms with hyperbolic tangent activation, with standard (top) vs normalized initialization (bottom). Top: 0-peak increases for higher layers.*

Figure 2

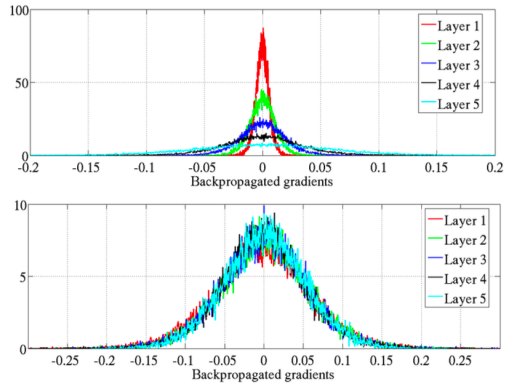


Figure 7: *Back-propagated gradients normalized histograms with hyperbolic tangent activation, with standard (top) vs normalized (bottom) initialization. Top: 0-peak decreases for higher layers.*

Table 1

Table 1: *Test error with different activation functions and initialization schemes for deep networks with 5 hidden layers. N after the activation function name indicates the use of normalized initialization. Results in bold are statistically different from non-bold ones under the null hypothesis test with* $p = 0.005$.

TYPE	Shapeset	MNIST	CIFAR-10	ImageNet
Softsign	**16.27**	**1.64**	55.78	**69.14**
Softsign N	**16.06**	**1.72**	**53.8**	**68.13**
Tanh	27.15	**1.76**	55.9	70.58
Tanh N	**15.60**	**1.64**	**52.92**	**68.57**
Sigmoid	82.61	2.21	57.28	70.66