Practical 10: Optimisers in AI Models

Need of Optimizer

In a neural network, an optimizer plays a crucial role in training the model. The primary purpose of an optimizer is to minimise the error between the predicted output of the neural network and the actual target output by adjusting the weights and biases of the network during the training process.

- Error Minimization: Neural networks are trained by adjusting their parameters (weights and biases) to minimise the difference between predicted outputs and actual targets. Optimizers determine how these parameters should be updated to reduce this error effectively.

- Gradient Descent: Most optimizers use some form of gradient descent algorithm to update the weights and biases of the neural network. Gradient descent computes the gradients of the loss function with respect to the network parameters, indicating the direction and magnitude of the steepest decrease in the error. By following these gradients, the optimizer updates the parameters to reach a minimum point in the error surface.

- Speed and Efficiency: Optimizers help in training the neural network efficiently by adjusting the learning rate and other hyperparameters. They prevent the model from getting stuck in local minima and help it converge to the global minimum of the loss function.

- Stability: Some optimizers incorporate techniques to stabilise the training process, such as momentum, adaptive learning rates, and regularisation. These techniques help prevent the model from oscillating or diverging during training.

- Flexibility: Different optimizers have different characteristics and are suitable for different types of problems or architectures. For instance, Adam optimizer is widely used for its adaptive learning rate properties, while SGD (Stochastic Gradient Descent) is simpler and may be more suitable for certain scenarios.

Overall, optimizers are essential components of neural network training, ensuring that the model learns effectively and efficiently from the training data to make accurate predictions on new data.

Significance of ADAM Optimizer

Adam (Adaptive Moment Estimation) optimizer is a popular optimization algorithm used in training neural networks. Its significance lies in several key features that make it effective for a wide range of deep learning tasks:

- Adaptive Learning Rates: Adam dynamically adjusts the learning rates for each parameter based on the magnitude of gradients and the past history of gradients for that parameter. This adaptivity helps in faster convergence and efficient training, as it allows larger updates for infrequent parameters and smaller updates for frequent ones.

- Momentum Optimization: Adam incorporates momentum, which helps accelerate convergence by accumulating gradients from past time steps. This helps to navigate through areas of high curvature and reach the minimum of the loss function more efficiently.

- Bias Correction: Adam performs bias correction, particularly in the early stages of training when the estimates of the moments are biased towards zero. This correction helps to improve the stability of training and ensures that the optimization process starts with more accurate estimates.

- Efficient Memory Usage: Adam maintains exponentially decaying averages of past gradients and squared gradients, requiring only first-order moments (mean) and second-order moments (variance). This results in efficient memory usage compared to other optimization algorithms like RMSprop.

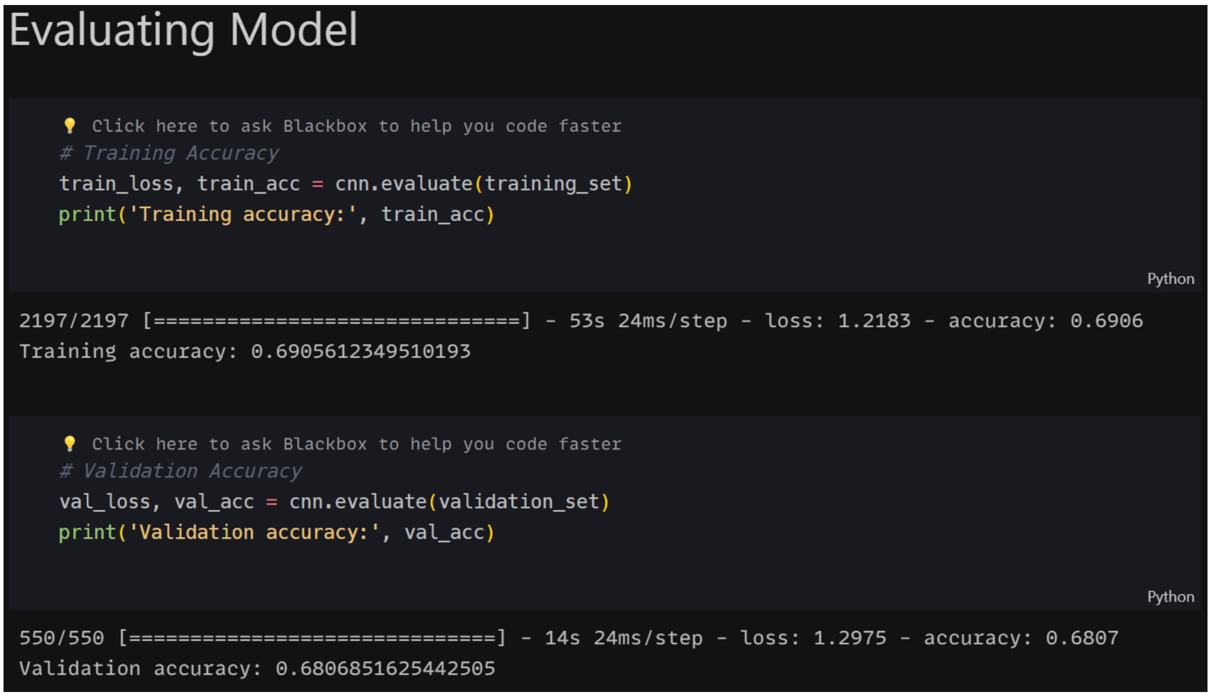
- Robustness to Hyperparameters: Adam is relatively less sensitive to hyperparameters compared to other optimization algorithms like SGD. It performs well with default hyperparameters across a wide range of tasks and architectures, making it easier to use for practitioners.

- Widely Used: Adam has become a standard optimizer in many deep learning frameworks and is often the default choice for training neural networks. Its widespread adoption and proven performance on various tasks make it a go-to optimizer for many practitioners.

Overall, the significance of Adam optimizer lies in its ability to provide efficient and effective optimization for neural networks, leading to faster convergence, better generalisation, and easier hyperparameter tuning.

Comparison with and without Optimizer

Without Optimiser



Evaluating Model

```
💡 Click here to ask Blackbox to help you code faster
# Training Accuracy
train_loss, train_acc = cnn.evaluate(training_set)
print('Training accuracy:', train_acc)
```

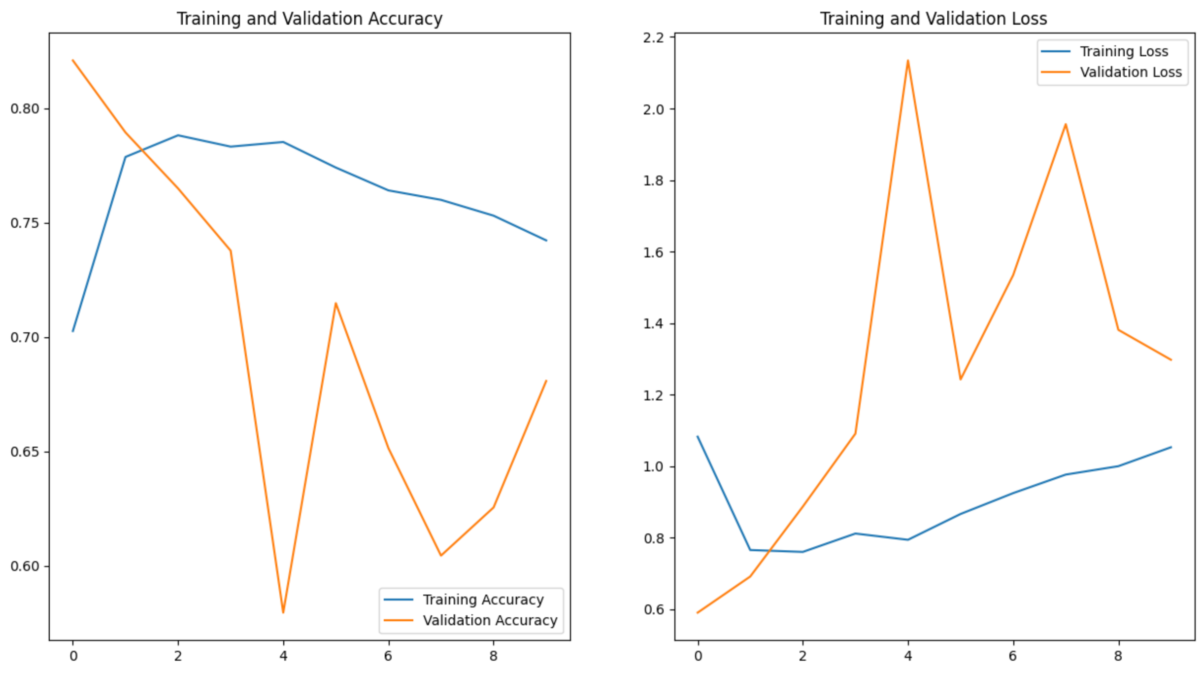
Python

```
2197/2197 [==============================] - 53s 24ms/step - loss: 1.2183 - accuracy: 0.6906
Training accuracy: 0.6905612349510193
```

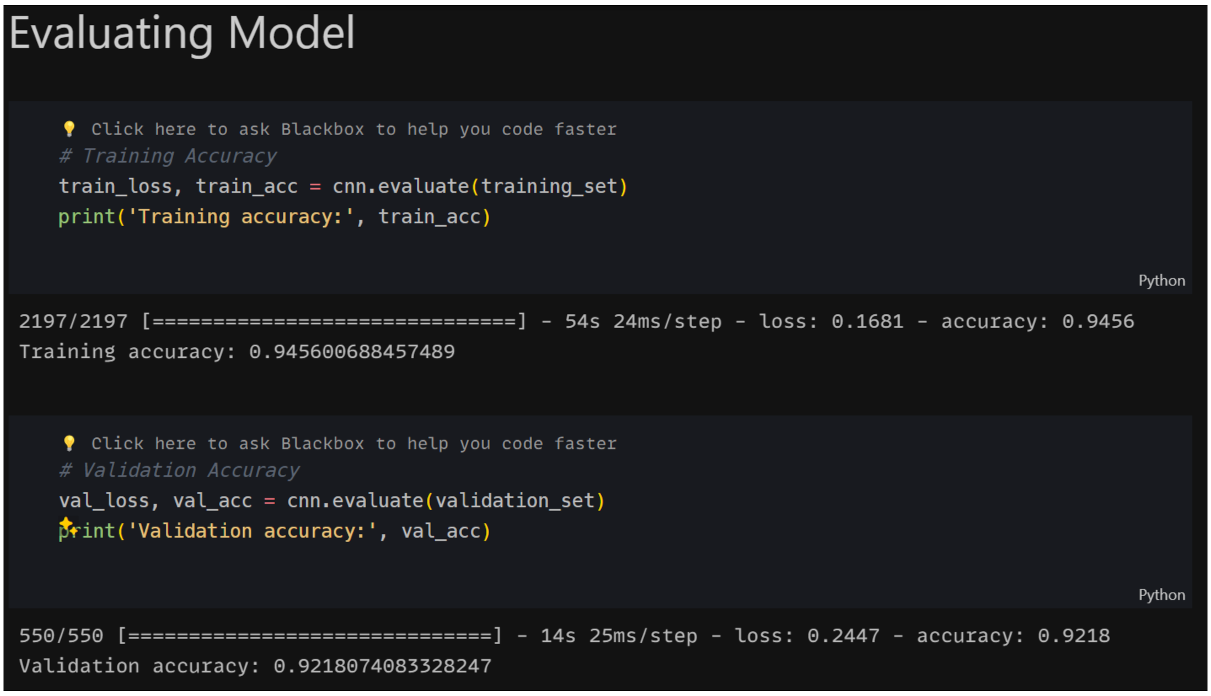
```
💡 Click here to ask Blackbox to help you code faster
# Validation Accuracy
val_loss, val_acc = cnn.evaluate(validation_set)
print('Validation accuracy:', val_acc)
```

Python

```
550/550 [==============================] - 14s 24ms/step - loss: 1.2975 - accuracy: 0.6807
Validation accuracy: 0.6806851625442505
```



With Optimiser



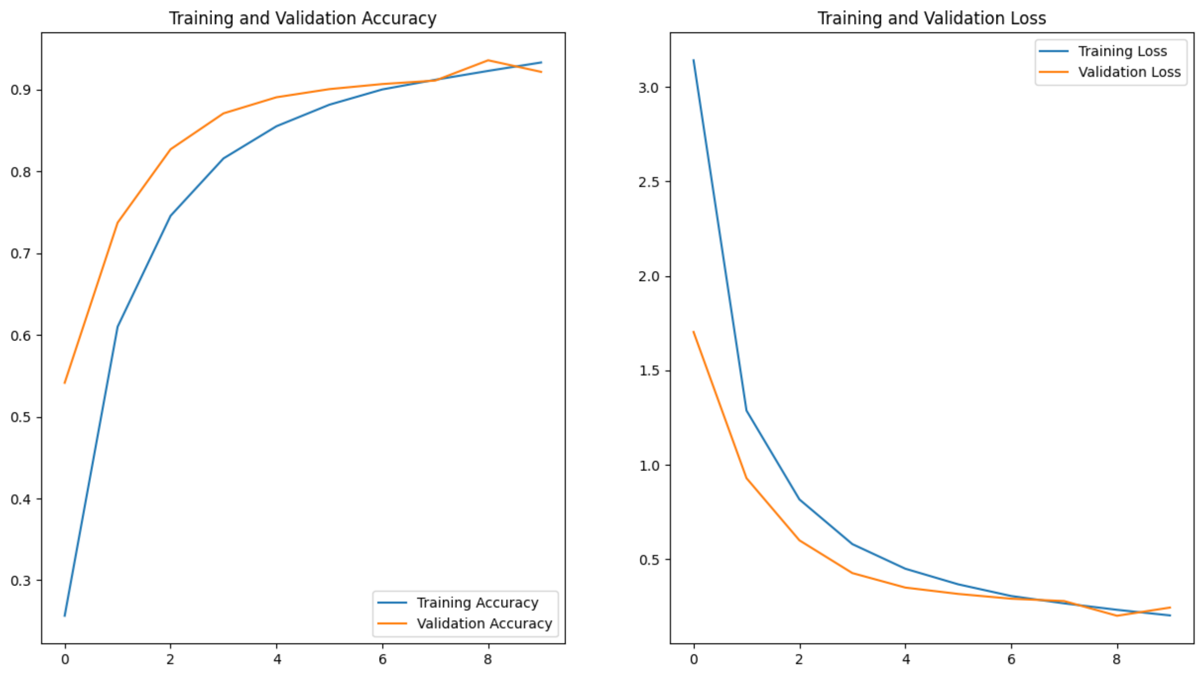
Evaluating Model

```
💡 Click here to ask Blackbox to help you code faster
# Training Accuracy
train_loss, train_acc = cnn.evaluate(training_set)
print('Training accuracy:', train_acc)
```

```
2197/2197 [==============================] - 54s 24ms/step - loss: 0.1681 - accuracy: 0.9456
Training accuracy: 0.945600688457489
```

```
💡 Click here to ask Blackbox to help you code faster
# Validation Accuracy
val_loss, val_acc = cnn.evaluate(validation_set)
print('Validation accuracy:', val_acc)
```

```
550/550 [==============================] - 14s 25ms/step - loss: 0.2447 - accuracy: 0.9218
Validation accuracy: 0.9218074083328247
```


Training and Validation Accuracy
Training Accuracy
Validation Accuracy
Training and Validation Loss
Training Loss
Validation Loss