Lesson 4: Training Deep Neural Networks

Training Deep Neural Networks is an iterative process that involves multiple steps, such as data preparation, network architecture selection, hyperparameter tuning, and optimization algorithm selection.

The first step in training a deep neural network is to prepare the input data. This may involve tasks such as cleaning the data, preprocessing it, and splitting it into training, validation, and testing sets.

Next, the network architecture is selected based on the nature of the task at hand. The architecture may involve different types of layers, such as convolutional, recurrent, and fully connected layers, and the number of layers and their sizes are chosen based on the complexity of the problem.

Once the network architecture is selected, the next step is to tune the hyperparameters of the network. Hyperparameters include the learning rate, regularization techniques, and batch size, among others. These hyperparameters can greatly affect the performance of the network and need to be tuned carefully to ensure that the network is able to learn from the data and generalize well to new inputs.

After the hyperparameters are tuned, the optimization algorithm is selected. Common optimization algorithms used in deep neural networks include stochastic gradient descent (SGD), Adam, and Adagrad, among others. The choice of optimization algorithm can greatly impact the convergence speed and performance of the network.

During the training process, the network is fed with input data, and the output is compared with the desired output to compute the loss. The optimizer algorithm is then used to update the network parameters in order to minimize the loss. This process is repeated over multiple iterations, or epochs, until the network converges to a satisfactory solution.

Overall, training deep neural networks is a complex and iterative process that requires careful selection of network architecture, hyperparameters, and optimization algorithms. With careful tuning, deep neural networks can achieve state-of-the-art performance on various tasks, making them a popular choice in many fields.

4.1 Regularization techniques (dropout, weight decay)

Regularization techniques are critical in deep learning to prevent overfitting, which is a common problem where the model becomes too complex and fits the training data too closely, leading to poor generalization to new data. Regularization techniques help to address this issue by adding constraints to the optimization process.

Dropout is a widely used regularization technique that randomly drops out a subset of neurons during training, forcing the remaining neurons to learn more robust and generalizable features. This technique has been shown to be effective in reducing overfitting in a variety of neural network architectures, including convolutional neural networks and recurrent neural networks.

Weight decay, also known as L2 regularization, is another popular regularization technique that adds a penalty term to the loss function to encourage the weights of the network to be small. This helps to prevent overfitting by reducing the complexity of the model and encouraging it to learn simpler representations of the data.

L1 regularization is a similar technique to weight decay, but it penalizes the absolute value of the weights instead of their squared magnitude. This can lead to sparse weight matrices, where some weights are set to zero, making the model more interpretable and reducing overfitting.

Early stopping is another regularization technique that involves monitoring the validation loss during training and stopping the training process when the validation loss starts to increase. This prevents the model from overfitting to the training data and helps to find a model that generalizes well to new data.

It's important to note that the choice of regularization technique depends on the specific problem and dataset, and there is ongoing research to develop new and more effective techniques.

4.2 Optimizers (SGD, Adam, Adagrad)

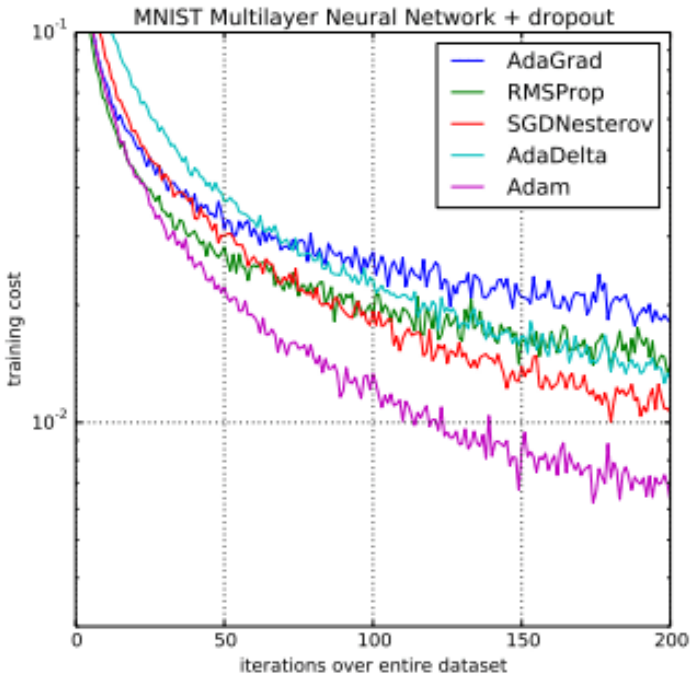
Optimizers play a crucial role in the training of deep neural networks by updating the parameters of the network to minimize the loss function. They help the network learn from the data and improve its predictions.

There are several optimizers commonly used in deep learning, each with its own strengths and weaknesses.

Stochastic Gradient Descent (SGD) is the most basic optimizer and is often used as a starting point for training deep neural networks. It works by computing the gradient of the loss function with respect to the parameters of the network for a small batch of data at a time. The parameters are then updated in the direction of the negative gradient, scaled by a learning rate hyperparameter. However, SGD can be slow to converge, especially for large datasets and complex models.

Adam is a popular optimizer that combines the ideas of momentum and adaptive learning rates. It uses a moving average of the gradients to adaptively scale the learning rate for each parameter. This helps the optimizer to converge faster and more reliably than SGD. Additionally, Adam has been shown to perform well in a wide range of applications, including computer vision and natural language processing.

Adagrad is another optimizer that adapts the learning rate for each parameter. However, instead of using a moving average of the gradients like Adam, Adagrad uses the sum of the squared gradients to compute an adaptive learning rate. This makes Adagrad well-suited for sparse data and non-convex optimization problems. However, Adagrad can sometimes become stuck in narrow valleys of the loss function.



Other popular optimizers include Adadelta, RMSprop, and Nadam. Adadelta and RMSprop are both adaptive learning rate optimizers that build on the ideas of Adagrad. Nadam is a combination of Adam and SGD with Nesterov momentum. Each optimizer has its own advantages and disadvantages, and the choice of optimizer should be based on the specific task and the properties of the data being used.

In addition to choosing the optimizer, the learning rate, batch size, and number of epochs are also important hyperparameters to tune during the training process. By selecting the right optimizer and tuning the hyperparameters appropriately, it is possible to achieve high performance on a wide range of deep learning tasks.

4.3 Learning rate schedules

Learning rate schedules are an important aspect of training deep neural networks. Learning rate is a hyperparameter that determines the step size at each iteration while moving toward a minimum of the loss function during training. It is a critical parameter that influences the performance and training time of deep neural networks.

A high learning rate can lead to divergent behavior, and the model may fail to converge or overshoot the optimal solution. On the other hand, a low learning rate can cause slow convergence and result in a longer training time.

One way to set the learning rate is to use a fixed value throughout the training process. However, a fixed learning rate may not always be optimal, as the optimal learning rate can change during training as the model approaches convergence. Therefore, learning rate schedules have been developed to adaptively change the learning rate during training.

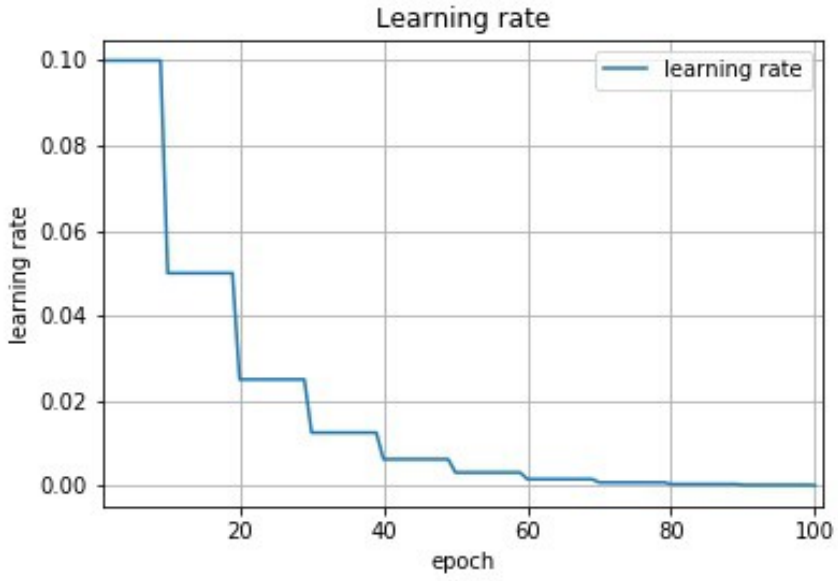
There are several types of learning rate schedules, including step decay, exponential decay, and polynomial decay. Step decay involves decreasing the learning rate by a factor after a fixed number of epochs. Exponential decay involves reducing the learning rate exponentially over time, while polynomial decay reduces the learning rate according to a polynomial function.

Another approach to setting the learning rate is to use adaptive learning rate methods, such as AdaGrad, RMSProp, and Adam. These methods dynamically adjust the learning rate based on the gradient of the loss function, allowing for faster convergence and better performance.

AdaGrad adapts the learning rate for each parameter based on its historical gradient. It assigns a smaller learning rate to the parameters that have larger gradient values and a higher learning rate to the parameters that have smaller gradient values. This technique is well-suited for sparse data and features with different scales.

RMSProp is a technique that also adapts the learning rate for each parameter. However, instead of using the historical gradient of the parameter, it uses the moving average of the squared gradient of the parameter to compute the adaptive learning rate. It helps to reduce the influence of noisy gradients in the parameter update.

Adam is a popular optimizer that combines the advantages of both AdaGrad and RMSProp. It uses a moving average of the gradient and the squared gradient of the parameter to compute the adaptive learning rate. Additionally, it incorporates momentum to speed up the convergence process and stabilize the updates.



Choosing the right learning rate schedule and optimizer is essential for training deep neural networks effectively. A well-chosen learning rate schedule can prevent the model from getting stuck in local minima and accelerate convergence. A suitable optimizer can improve performance and reduce training time, leading to better and faster results.

4.4 CODE EXAMPLE

Regularization techniques (dropout, weight decay):

In neural network models, regularization techniques are used to prevent overfitting, which occurs when a model becomes too complex and starts to fit the noise in the data rather than the underlying patterns. Dropout and weight decay are two common regularization techniques.

Example code for implementing dropout in PyTorch:

```
import torch
import torch.nn as nn


class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(10, 50)
        self.dropout = nn.Dropout(p=0.5)
        self.fc2 = nn.Linear(50, 1)


    def forward(self, x):
        x = self.fc1(x)
        x = nn.functional.relu(x)
        x = self.dropout(x)
        x = self.fc2(x)
        return x
```

Explanation: In this example, we define a neural network model with a fully connected layer (fc1) that takes in an input of size 10 and outputs a hidden layer of size 50. We then apply a dropout layer with a dropout probability of 0.5, which randomly sets half of the activations in the hidden layer to zero during each forward pass. Finally, we have another fully connected layer (fc2) that takes in the output of the dropout layer and produces a single output.

Optimizers (SGD, Adam, Adagrad):

In order to update the weights of a neural network during training, we need to use an optimizer that minimizes the loss function. Stochastic gradient descent (SGD), Adam, and Adagrad are three common optimizers used in deep learning.

Example code for implementing Adam optimizer in TensorFlow:

```
import tensorflow as tf



# Define the model
model = tf.keras.Sequential([
    tf.keras.layers.Dense(64, activation='relu', input_shape=(784,)),
    tf.keras.layers.Dense(10, activation='softmax')
])



# Compile the model with Adam optimizer
model.compile(optimizer='adam',
              loss='categorical_crossentropy',
              metrics=['accuracy'])
```

Explanation: In this example, we define a neural network model with two fully connected layers, where the first layer has 64 units and uses the ReLU activation function. We then compile the model with the Adam optimizer, which is a popular optimizer that adapts the learning rate based on the momentum and the root mean square of the gradients. We also specify the loss function as categorical crossentropy and the evaluation metric as accuracy.

Learning rate schedules:

The learning rate is a hyperparameter that controls how quickly the model learns from the data. Learning rate schedules are used to adjust the learning rate over time to improve convergence.

Example code for implementing a learning rate schedule in TensorFlow:

```
import tensorflow as tf
from tensorflow.keras.datasets import mnist
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.callbacks import LearningRateScheduler


# Load the MNIST dataset
(x_train, y_train), (x_test, y_test) = mnist.load_data()


# Normalize pixel values to be between 0 and 1
x_train = x_train / 255.0
x_test = x_test / 255.0


# Convert class vectors to binary class matrices
y_train = tf.keras.utils.to_categorical(y_train, num_classes=10)
y_test = tf.keras.utils.to_categorical(y_test, num_classes=10)


# Define a function to create the model
def create_model():
    model = Sequential()
    model.add(Dense(512, activation='relu', input_shape=(784,)))
    model.add(Dense(10, activation='softmax'))
    return model


# Define a function to create the learning rate schedule
def lr_schedule(epoch):
```

```
    lr = 0.1
    if epoch > 10:
        lr = 0.01
    if epoch > 20:
        lr = 0.001
    return lr


# Create the model and compile it
model = create_model()
model.compile(loss='categorical_crossentropy', optimizer='sgd',
metrics=['accuracy'])


# Create the learning rate scheduler callback
lr_scheduler = LearningRateScheduler(lr_schedule)


# Train the model with the learning rate scheduler
model.fit(x_train, y_train, batch_size=128, epochs=30,
callbacks=[lr_scheduler], validation_data=(x_test, y_test))
```

Explanation: In this example, we define a function **lr_schedule** that takes the current epoch as an argument and returns the learning rate for that epoch. The learning rate starts at 0.1 and is reduced to 0.01 after 10 epochs and 0.001 after 20 epochs.

We then create a **LearningRateScheduler** callback and pass it to the fit method of the model. This callback will be called at the beginning of each epoch and will set the learning rate for that epoch based on the **lr_schedule** function.
