notes_28

January 27, 2025

1 Previous Class Definitions

[1]:
```
# imports
import matplotlib.pyplot as plt
import numpy as np
import nnfs
from nnfs.datasets import spiral_data, vertical_data
nnfs.init()
```

[2]:
```
class Layer_Dense:
    def __init__(self, n_inputs, n_neurons):
        # Initialize the weights and biases
        self.weights = 0.01 * np.random.randn(n_inputs, n_neurons)  # Normal␣
 ↪distribution of weights
        self.biases = np.zeros((1, n_neurons))

    def forward(self, inputs):
        # Calculate the output values from inputs, weights, and biases
        self.inputs = inputs
        self.output = np.dot(inputs, self.weights) + self.biases        #␣
 ↪Weights are already transposed

    def backward(self, dvalues):
        '''Calculated the gradient of the loss with respect to the weights and␣
 ↪biases of this layer.
        dvalues is equiavelent to a transposed dl_dZ. It is the gradient
        of the loss with respect to the outputs of this layer.'''
        self.dweights = np.dot(self.inputs.T, dvalues)
        self.dbiases = np.sum(dvalues, axis=0, keepdims=0)
        self.dinputs = np.dot(dvalues, self.weights.T)

class Activation_ReLU:
    def forward(self, inputs):
        self.inputs = inputs
        self.output = np.maximum(0, inputs)

    def backward(self, dvalues):
```

```
        '''Calculated the gradient of the loss with respect to this layer's␣
 ↪activation function
        dvalues is equiavelent to a transposed dl_dZ. It is the gradient
        of the loss with respect to the outputs of this layer.'''
        self.dinputs = dvalues.copy()
        self.dinputs[self.inputs <= 0] = 0

class Activation_Softmax:
    def forward(self, inputs):
        # Get the unnormalized probabilities
        # Subtract max from the row to prevent larger numbers
        exp_values = np.exp(inputs - np.max(inputs, axis=1, keepdims=True))

        # Normalize the probabilities with element wise division
        probabilities = exp_values / np.sum(exp_values, axis=1,keepdims=True)
        self.output = probabilities

# Base class for Loss functions
class Loss:
    '''Calculates the data and regularization losses given
    model output and ground truth values'''
    def calculate(self, output, y):
        sample_losses = self.forward(output, y)
        data_loss = np.average(sample_losses)
        return data_loss

class Loss_CategoricalCrossEntropy(Loss):
    def forward(self, y_pred, y_true):
        '''y_pred is the neural network output
        y_true is the ideal output of the neural network'''
        samples = len(y_pred)
        # Bound the predicted values
        y_pred_clipped = np.clip(y_pred, 1e-7, 1-1e-7)

        if len(y_true.shape) == 1:      # Categorically labeled
            correct_confidences = y_pred_clipped[range(samples), y_true]
        elif len(y_true.shape) == 2:    # One hot encoded
            correct_confidences = np.sum(y_pred_clipped*y_true, axis=1)

        # Calculate the losses
        negative_log_likelihoods = -np.log(correct_confidences)
        return negative_log_likelihoods

    def backward(self, dvalues, y_true):
        samples = len(dvalues)

        # Number of lables in each sample
```

```
        labels = len(dvalues[0])

        # if the labels are sparse, turn them into a one-hot vector
        if len(y_true.shape) == 1:
            y_true = np.eye(labels)[y_true]

        # Calculate the gradient then normalize
        self.dinputs = -y_true / dvalues
        self.dinputs = self.dinputs / samples

class Activation_Softmax_Loss_CategoricalCrossentropy():
    def __init__(self):
        self.activation = Activation_Softmax()
        self.loss = Loss_CategoricalCrossEntropy()

    def forward(self, inputs, y_true):
        self.activation.forward(inputs)
        self.output = self.activation.output
        return self.loss.calculate(self.output, y_true)

    def backward(self, dvalues, y_true):
        samples = len(dvalues)

        # if the samples are one-hot encoded, turn them into discrete values
        if len(y_true.shape) == 2:
            y_true = np.argmax(y_true, axis=1)

        # Copy so we can safely modify
        self.dinputs = dvalues.copy()

        # Calculate and normalize gradient
        self.dinputs[range(samples), y_true] -= 1
        self.dinputs = self.dinputs / samples

class Optimizer_SGD():
    def __init__(self, learning_rate=0.5, decay=0.0, momentum=0.0):
        self.initial_rate = learning_rate
        self.current_learning_rate = self.initial_rate
        self.decay = decay
        self.iterations = 0
        self.momentum = momentum

    def pre_update_params(self):
        # Update the current_learning_rate before updating params
        if self.decay:
            self.current_learning_rate = self.initial_rate / (1 + self.decay *␣
 ↪self.iterations)
```

```
    def update_params(self, layer):
        if self.momentum:
            # For each layer, we need to use its last momentums

            # First check if the layer has a last momentum stored
            if not hasattr(layer, 'weight_momentums'):
                layer.weight_momentums = np.zeros_like(layer.weights)
                layer.bias_momentums = np.zeros_like(layer.biases)

            weight_updates = self.momentum * layer.weight_momentums - \
                             self.current_learning_rate * layer.dweights
            layer.weight_momentums = weight_updates

            bias_updates = self.momentum * layer.bias_momentums - \
                           self.current_learning_rate * layer.dbiases
            layer.bias_momentums = bias_updates

        # Not using momentum
        else:
            weight_updates = -self.current_learning_rate * layer.dweights
            bias_updates = -self.current_learning_rate * layer.dbiases

        layer.weights += weight_updates
        layer.biases += bias_updates

    def post_update_params(self):
        # Update the self.iterations for use with decay
        self.iterations += 1

class Optimizer_Adagrad():
    def __init__(self, learning_rate=0.5, decay=0.0, epsilon=1e-7):
        self.initial_learning_rate = learning_rate
        self.current_learning_rate = self.initial_learning_rate
        self.decay = decay
        self.iterations = 0
        self.epsilon = epsilon

    def pre_update_params(self):
        if self.decay:
            self.current_learning_rate = self.initial_learning_rate / (1 + self.
 ↪decay * self.iterations)

    def update_params(self, layer):
        if not hasattr(layer, 'weight_cache'):
            layer.weight_cache = np.zeros_like(layer.weights)
            layer.bias_cache = np.zeros_like(layer.biases)
```

```
        layer.weight_cache += layer.dweights**2
        layer.bias_cache += layer.dbiases**2

        layer.weights += -self.current_learning_rate * layer.dweights / (np.
 ↪sqrt(layer.weight_cache) + self.epsilon)
        layer.biases += -self.current_learning_rate * layer.dbiases / (np.
 ↪sqrt(layer.bias_cache) + self.epsilon)

    def post_update_params(self):
        self.iterations += 1

class Optimizer_RMSProp():
    def __init__(self, learning_rate=1e-3, decay=0.0, epsilon=1e-7, rho=0.9):
        self.initial_learning_rate = learning_rate
        self.current_learning_rate = self.initial_learning_rate
        self.decay = decay
        self.iterations = 0
        self.epsilon = epsilon
        self.rho = rho

    def pre_update_params(self):
        if self.decay:
            self.current_learning_rate = self.initial_learning_rate / (1 + self.
 ↪decay * self.iterations)

    def update_params(self, layer):
        if not hasattr(layer, 'weight_cache'):
            layer.weight_cache = np.zeros_like(layer.weights)
            layer.bias_cache = np.zeros_like(layer.biases)

        layer.weight_cache = self.rho * layer.weight_cache + (1 - self.rho) *␣
 ↪layer.dweights**2
        layer.bias_cache = self.rho * layer.bias_cache + (1 - self.rho) * layer.
 ↪dbiases**2

        layer.weights += -self.current_learning_rate * layer.dweights / (np.
 ↪sqrt(layer.weight_cache) + self.epsilon)
        layer.biases += -self.current_learning_rate * layer.dbiases / (np.
 ↪sqrt(layer.bias_cache) + self.epsilon)

    def post_update_params(self):
        self.iterations += 1

# Adam optimizer
class Optimizer_Adam():
```

```
  def __init__(self, learning_rate=0.001, decay=0.0, epsilon=1e-7, beta_1=0.
↪9, beta_2=0.999):
      self.initial_learning_rate = learning_rate
      self.current_learning_rate = learning_rate
      self.decay = decay
      self.iterations = 0
      self.epsilon = epsilon
      self.beta_1 = beta_1
      self.beta_2 = beta_2

  def pre_update_params(self):
      if self.decay:
          self.current_learning_rate = self.initial_learning_rate * (1. / (1.␣
↪+ self.decay * self.iterations))

  def update_params(self, layer):
      # If layer does not contain cache arrays, create them filled with zeros
      if not hasattr(layer, 'weight_cache'):
          layer.weight_momentums = np.zeros_like(layer.weights)
          layer.weight_cache = np.zeros_like(layer.weights)
          layer.bias_momentums = np.zeros_like(layer.biases)
          layer.bias_cache = np.zeros_like(layer.biases)

      # Update momentum with current gradients
      layer.weight_momentums = self.beta_1 * layer.weight_momentums + (1 -␣
↪self.beta_1) * layer.dweights
      layer.bias_momentums = self.beta_1 * layer.bias_momentums + (1 - self.
↪beta_1) * layer.dbiases

      # Get corrected momentum
      # use self.iteration + 1 because we start at iteration 0
      weight_momentums_corrected = layer.weight_momentums / (1 - self.beta_1␣
↪** (self.iterations + 1))
      bias_momentums_corrected = layer.bias_momentums / (1 - self.beta_1 **␣
↪(self.iterations + 1))

      # Update cache with squared current gradients
      layer.weight_cache = self.beta_2 * layer.weight_cache + (1 - self.
↪beta_2) * layer.dweights**2
      layer.bias_cache = self.beta_2 * layer.bias_cache + (1 - self.beta_2) *␣
↪layer.dbiases**2

      # Get corrected cache
      weight_cache_corrected = layer.weight_cache / (1 - self.beta_2 ** (self.
↪iterations + 1))
```

```
        bias_cache_corrected = layer.bias_cache / (1 - self.beta_2 ** (self.
↪iterations + 1))

        # Vanilla SGD parameter update + normalization with square rooted cache
        layer.weights += -self.current_learning_rate *␣
↪weight_momentums_corrected / (np.sqrt(weight_cache_corrected) + self.epsilon)
        layer.biases += -self.current_learning_rate * bias_momentums_corrected /
↪ (np.sqrt(bias_cache_corrected) + self.epsilon)

    # Call once after any parameter updates
    def post_update_params(self):
        self.iterations += 1
```

2 Generalization and Overfitting

Overfitting can occur when the neural network tries to fit every training data perfectly. If the training data was perfect, this would not be an issue. However, because some training data is bad or should not be expected to be identified perfectly, the neural network can sacrifice generability to trying to identify all training data.

If we could assign uncertainty to training data, I believe this would help.

2.1 Out of Sample Data

Rather than use all of our data for training, we can set aside some for out of sample testing so we can better understand how well the network generalizes.

2.1.1 Run the Adam Optimizer with the 100 Samples of Training Data

[4]:
```
# Create dataset
X, y = spiral_data(samples=100, classes=3)

# Create Dense layer with 2 input features and 64 output values
dense1 = Layer_Dense(2, 64)

# Create ReLU activation (to be used with Dense layer)
activation1 = Activation_ReLU()

# Create second Dense layer with 64 input features (as we take output
# of previous layer here) and 3 output values (output values)
dense2 = Layer_Dense(64, 3)

# Create Softmax classifier's combined loss and activation
loss_activation = Activation_Softmax_Loss_CategoricalCrossentropy()

# Create optimizer
optimizer = Optimizer_Adam(learning_rate=0.02, decay=1e-5)
```

```
# Train in loop
for epoch in range(10001):
    # Perform a forward pass of our training data through this layer
    dense1.forward(X)

    # Perform a forward pass through activation function
    # takes the output of first dense layer here
    activation1.forward(dense1.output)

    # Perform a forward pass through second Dense layer
    # takes outputs of activation function of first layer as inputs
    dense2.forward(activation1.output)

    # Perform a forward pass through the activation/loss function
    # takes the output of second dense layer here and returns loss
    loss = loss_activation.forward(dense2.output, y)

    # Calculate accuracy from output of activation2 and targets
    # calculate values along first axis
    predictions = np.argmax(loss_activation.output, axis=1)
    if len(y.shape) == 2:
        y = np.argmax(y, axis=1)
    accuracy = np.mean(predictions == y)

    # Backward pass
    loss_activation.backward(loss_activation.output, y)
    dense2.backward(loss_activation.dinputs)
    activation1.backward(dense2.dinputs)
    dense1.backward(activation1.dinputs)

    # Update weights and biases
    optimizer.pre_update_params()
    optimizer.update_params(dense1)
    optimizer.update_params(dense2)
    optimizer.post_update_params()

print(f'epoch: {epoch}, ' +
      f'acc: {accuracy:.3f}, ' +
      f'loss: {loss:.3f}, ' +
      f'lr: {optimizer.current_learning_rate}')
```

```
epoch: 10000, acc: 0.883, loss: 0.230, lr: 0.01818181818181818
```

2.1.2 Now Use Different Found Biases and Weights on Out of Sample Data

[5]:
```
# Create test dataset
X_test, y_test = spiral_data(samples=100, classes=3)
# Perform a forward pass of our testing data through this layer
dense1.forward(X_test)
# Perform a forward pass through activation function
# takes the output of first dense layer here
activation1.forward(dense1.output)
# Perform a forward pass through second Dense layer
# takes outputs of activation function of first layer as inputs
dense2.forward(activation1.output)
# Perform a forward pass through the activation/loss function
# takes the output of second dense layer here and returns loss
loss = loss_activation.forward(dense2.output, y_test)
# Calculate accuracy from output of activation2 and targets
# calculate values along first axis
predictions = np.argmax(loss_activation.output, axis=1)
if len(y_test.shape) == 2:
 y_test = np.argmax(y_test, axis=1)
accuracy = np.mean(predictions == y_test)
print(f'validation, acc: {accuracy:.3f}, loss: {loss:.3f}')
```

```
validation, acc: 0.797, loss: 0.672
```

2.1.3 Observations

The out of sample accuracy is about 0.1% lower than the training data, with a loss almost 3x the training data.

3 Preventing Overfitting

3.1 Reducing Network Complexity

Simpler models are more robust against overfitting and can provide more generalizability. This can be reducing the number of neurons in a layer or the total layers. Effectively, you reduce the granularity of functions that the network can model.

3.2 Reduce the Number of Epochs

By allowing less training iterations to occur, the network isn't given the time or opportunity to fit data points that might not be valid.

These "hyper-parameters" can be adjusted after testing with out of sample data.

4 Types of Data

Training data is used to optimize a given network and minimize loss.

Validation data is used to optimize hyper-parameters of the network. Parameters like number of layers, neurons, neurons per layer, activation layer and their constants, epochs, learning rates, etc.

Testing data is used to see the out of sample effectiveness of the trained network.

4.1 Splitting up Data

4.1.1 Given a Lot of Data

Dataset is broken primarily into training data and then also validation and testing data.

4.1.2 Given Limited Data

Dataset is only broken up into training data and testing data (maybe 80%-20%). K-Fold validation can be used in limited dataset scenarios.

5 K-Fold Cross Validation

In the limited training dataset, you can split the dataset further into subsections, say 5. You then have 5 different combinations of data, where 1 subsection is considered the validation while the other 4 are considered the training data.

When using 5 subsections of the training data, say {A, B, C, D, E}, you get 5 validation losses. The total validation loss is considered the average of all 5.

For determining different hyper-parameters, you run the network on the same training data and choose the one with the lowest total validation loss.

5.1 Data Leakage

While K-Fold is good for getting hyper-parameters with limited data, it can have data leakage if not correctly setup. For example, with timeseries data, it may get access to future information and train off of that.