Constructing Custom Logit Function with pretrained BERT model

Compiled by D.Gueorguiev, 5/5/2025

To construct a custom logit function for BERT, you'll typically define a new output layer on top of the pre-trained BERT model. This new layer will take BERT's output and map it to the logits for your specific task. This involves defining a linear layer (using libraries like PyTorch or TensorFlow) and applying it to BERT's final hidden state.

Here's a more detailed breakdown:

1. **1. Load the pre-trained BERT model:**

You'll use a library like Hugging Face Transformers to load a pre-trained BERT model.

1. **2. Define the custom output layer:**
   * **Linear layer:** Create a linear (or fully connected) layer with the number of input neurons corresponding to BERT's output dimension and the number of output neurons equal to the number of classes in your task.
   * **Apply the layer:** In the forward pass of your model, take the final hidden state from BERT (usually pooler\_output or the last hidden state) and apply the linear layer to it.
2. **3. Forward pass:**
   * In your model's forward function, include the BERT model's output as input.
   * Pass the output of BERT's encoder to your custom output layer.
   * Return the logits produced by the linear layer.
3. **4. Training:**
   * Use a loss function appropriate for your task (e.g., CrossEntropyLoss for classification).
   * Train the model, optimizing for the custom output layer and potentially fine-tuning BERT's weights depending on your needs.

Key Considerations:

* **[Output Dimension:](https://www.google.com/search?rlz=1C5GCEM_enUS1034US1034&cs=0&sca_esv=8e26205c2bafb5f9&sxsrf=AHTn8zoHp7GNV1fC5CzfP541INQ0kjSIqg%3A1746471067134&q=Output+Dimension&sa=X&ved=2ahUKEwjOrNHb_4yNAxX-FmIAHZaqHO0Q4eYNegQIRRAD&mstk=AUtExfBdN5EsqIgS9DOXMGP1FiAPzdwPqRF2TM0yMpuu0c-kT04BLtIwtdnwhiZS3wALDDz-gxNhKBuR2Gzi9hqURLLp6vkEDlFPCotSCXZBdhKAxcca-2dFBZw3LJf_lH_vOp_vX-YrBS7p1j3Yhq5bbGPwAEphH_HxMKeQ9abZSGgBYw8&csui=3" \t "_blank)**

The number of output neurons in your custom layer needs to match the number of classes or the target variable in your task.

* **Fine-tuning vs. Static Weights**:

You can choose to fine-tune BERT's weights during training, or keep them fixed, depending on your task and computational resources.

* **Loss Function**:

Select an appropriate loss function (e.g., CrossEntropyLoss for classification, MSE for regression).

* [**Optimization:**](https://www.google.com/search?rlz=1C5GCEM_enUS1034US1034&cs=0&sca_esv=8e26205c2bafb5f9&sxsrf=AHTn8zoHp7GNV1fC5CzfP541INQ0kjSIqg%3A1746471067134&q=Optimization&sa=X&ved=2ahUKEwjOrNHb_4yNAxX-FmIAHZaqHO0Q4eYNegQIWBAD&mstk=AUtExfBdN5EsqIgS9DOXMGP1FiAPzdwPqRF2TM0yMpuu0c-kT04BLtIwtdnwhiZS3wALDDz-gxNhKBuR2Gzi9hqURLLp6vkEDlFPCotSCXZBdhKAxcca-2dFBZw3LJf_lH_vOp_vX-YrBS7p1j3Yhq5bbGPwAEphH_HxMKeQ9abZSGgBYw8&csui=3)

Choose an appropriate optimizer (e.g., Adam) and adjust learning rates accordingly.

By following these steps, you can effectively create custom logit functions for BERT to adapt it to various tasks and achieve desired performance.

PyTorch Example:

import torch

from transformers import BertModel, BertConfig, BertTokenizer

from torch import nn

class CustomBertClassifier(nn.Module):

def \_\_init\_\_(self, bert\_model\_name, num\_classes):

super(CustomBertClassifier, self).\_\_init\_\_()

self.bert = BertModel.from\_pretrained(bert\_model\_name) # Load pre-trained BERT

self.classifier = nn.Linear(self.bert.config.hidden\_size, num\_classes) # Custom output layer

def forward(self, input\_ids, attention\_mask):

# BERT output: [batch\_size, sequence\_length, hidden\_size]

outputs = self.bert(input\_ids, attention\_mask=attention\_mask)

pooled\_output = outputs[1] # Use pooled output (CLS token)

logits = self.classifier(pooled\_output) # Apply custom classifier

return logits