

Business 4720 - Class 15

Neural Networks using Python

Joerg Evermann

Faculty of Business Administration
Memorial University of Newfoundland
`jevermann@mun.ca`



Unless otherwise indicated, the copyright in this material is owned by Joerg Evermann. This material is licensed to you under the [Creative Commons by-attribution non-commercial license \(CC BY-NC 4.0\)](#)

What You Will Learn:

- ▶ Deep Learning Concepts
 - ▶ Neural Network
 - ▶ Activation Functions
 - ▶ Gradients
 - ▶ Backpropagation
 - ▶ Regularization with Dropouts
- ▶ Deep Learning in Python using Tensorflow
 - ▶ Tensors
 - ▶ Models
 - ▶ Training

Based On

Gareth James, Daniel Witten, Trevor Hastie and Robert Tibshirani:
An Introduction to Statistical Learning with Applications in R. 2nd
edition, corrected printing, June 2023. (ISLR2)

<https://www.statlearning.com>

Chapter 10

Kevin P. Murphy: *Probabilistic Machine Learning – An Introduction*.
MIT Press 2022.

<https://probml.github.io/pml-book/book1.html>

Chapter 13, 14, 15

Tensorflow Guides

<https://www.tensorflow.org/guide>

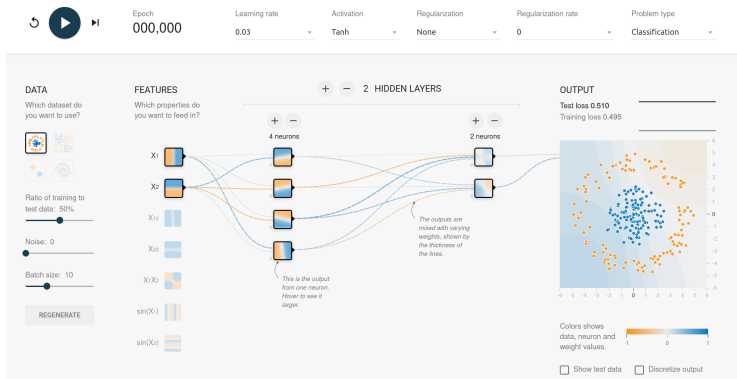
Implementations are available on the following GitHub repo:

`https://github.com/jevermann/busi4720-ml`

The project can be cloned from this URL:

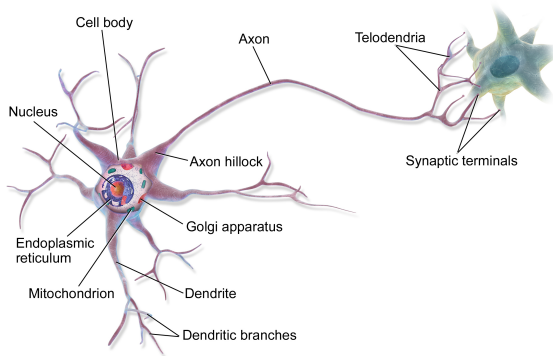
`https://github.com/jevermann/busi4720-ml.git`

Tensorflow Playground: <https://playground.tensorflow.org>



Biological Neuron

- ▶ Brain cell
- ▶ Connected to other brain cells
- ▶ Receives, modulates and emits electro-chemical stimulus ("activation")



https://commons.wikimedia.org/wiki/File:Blausen_0657_MultipolarNeuron.png

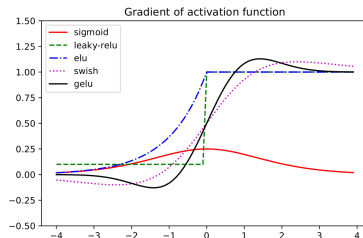
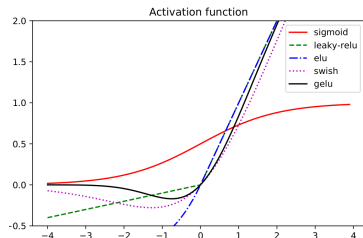
Artificial Neuron

$$y = \psi(b + \sum_i w_i x_i)$$

- ▶ Multiple **input** connections x_i
- ▶ Weighted using **weights** w_i
- ▶ Add a **bias** term b
- ▶ Apply *nonlinear* **activation function** ψ

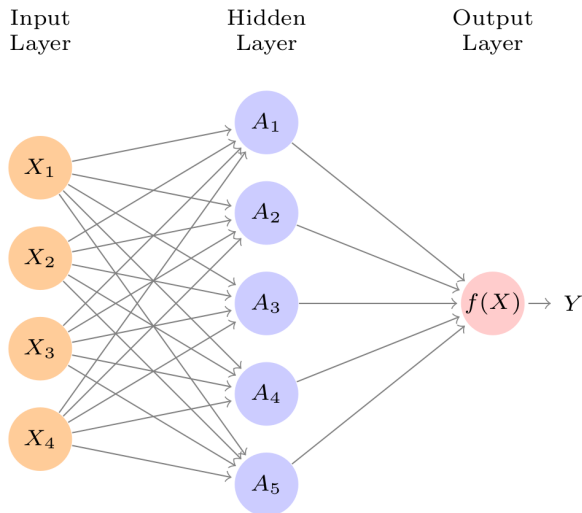
Popular Activation Functions

Sigmoid	$\sigma(z) = \frac{e^z}{1+e^z}$
Hyperbolic tangent	$\tanh(z) = \frac{\sinh(z)}{\cosh(z)} = \frac{e^z - e^{-z}}{e^z + e^{-z}} = 2\sigma(2z) - 1$
Softplus	$\sigma_+(z) = \log(1 + e^z)$
Rectified linear unit	$\text{ReLU}(z) = \max(z, 0)$
Leaky ReLU	$\text{LReLU}(z) = \max(z, 0) + \alpha \min(z, 0)$
Exponential linear unit	$\text{ELU}(z) = \max(z, 0) + \min(\alpha(e^z - 1), 0)$
Swish, Sigmoid linear unit	$\text{SiLU}(z) = z\sigma(z)$
Gaussian error linear unit	$\text{GeLU}(z) = z\Phi(z)$



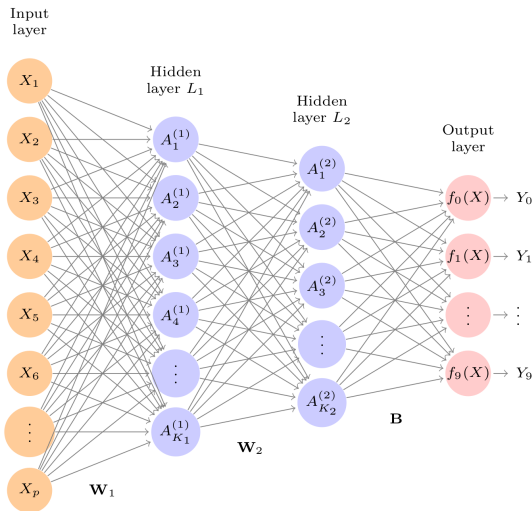
Source:
Murphy,
Fig.
13.14

Fully Connected Hidden Layer



Source: ISLR2 Figure 10.1

Fully Connected Multilayer Network



Source: ISLR2 Figure 10.4

Multiple Outputs
either

- Multi-objective learning
- Multi-class classification

"Softmax" activation

$$\Pr(Y = m|X) = \frac{e^{z_m}}{\sum_{l=0}^n e^{z_l}}$$

Estimating Parameters

Typical Loss Functions

- ▶ **Regression:** MSE, MAE, Huber
- ▶ **Classification:** Cross-Entropy or KL-Divergence after softmax on multiple output nodes

Parameters

- ▶ Parameter vector $\theta = (w, b)$ with weights w and biases b .

Optimization

- ▶ (Stochastic) gradient descent (SGD)

Regularization

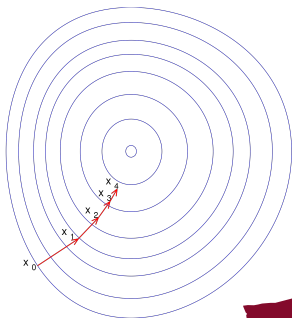
- ▶ "Dropout"
- ▶ L1 and/or L2 penalization (as in lasso, ridge)
- ▶ Early stopping

Gradient Descent

- 1 Begin with initial parameter values
- 2 Repeat until convergence
 - 2.1 Find direction of descent (decrease in loss function value, given by the gradient vector ∇L of partial derivatives)
 - 2.2 Move a step in that direction (adjust parameters, step size determined by **learning rate**)

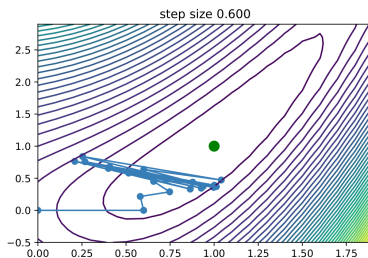
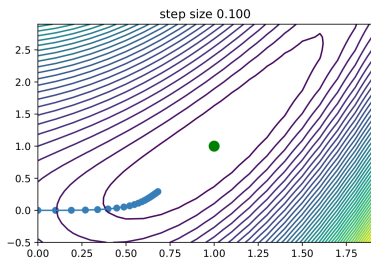
Consider the loss L at a certain input X as a function of parameter values θ . Then, at each step t , update parameters θ using learning rate γ :

$$\theta_{t+1} = \theta_t - \gamma \nabla L(\theta)|_{\theta_t, X}$$



Optimization Problems

- ▶ Slow convergence
- ▶ No convergence (oscillations)
- ▶ Premature convergence (local optimum)



Murphy, Figure 8.11

Learning Rate

- ▶ Fixed step size
- ▶ Adaptive learning rate λ_t
- ▶ Momentum methods
- ▶ Optimal learning rate ("line search")

Training Neural Networks – Epochs and Minibatches

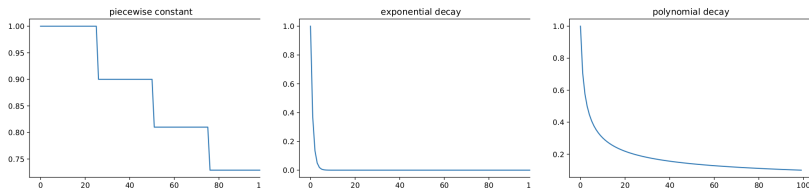
- ▶ Using the full training set for every update step is expensive (or impossible)
- ▶ Approximate true gradient by using a small sample of the training set for each step, the **minibatch**
 - ▶ Average gradients over minibatch
 - ▶ Minibatch should be independent and random
 - ▶ Minibatch size should not be "too small"
- ▶ Multiple passes over the training set until convergence (a local or global optimum is found), the **epochs**
 - ▶ Avoid repetition by shuffling the training set before each epoch
 - ▶ Early stopping for regularization

Optimization Options – Stochastic Gradient Descent

- ▶ Random inputs X to gradients by random draws from training set

$$\theta_{t+1} = \theta_t - \lambda_t \nabla L(\theta)|_{\theta_t, X}$$

- ▶ Requires adaptive learning rate
- ▶ Typical learning rate schedules: Piecewise constant, exponential decay, polynomial decay



Source: Murphy Figure 8.18

Optimization Options – AdaGrad

Adaptive Gradient

- ▶ Originally developed for sparse gradient vectors
- ▶ Adapt by previous squared gradients
- ▶ Overall learning rate λ_t is adapted
- ▶ Typically: $\lambda_t = \lambda_0$

$$\theta_{t+1} = \theta_t - \lambda_t \frac{1}{\sqrt{s_t + \epsilon}} \nabla L(\theta)|_{\theta_t, X}$$

$$s_t = \sum_{\tau=1}^t (\nabla L(\theta)|_{\theta_\tau, X})^2 \quad \text{Sum of squared gradients}$$

- ▶ Exponentially weighted moving average of the past (instead of the sum as in AdaGrad)
- ▶ Prevents too early learning rate reduction

$$\mathbf{s}_{t+1} = \beta \mathbf{s}_t + (1 - \beta) (\nabla L(\theta)|_{\theta_t, X})^2$$

- Maintains exponentially weighted average of previous updates δ

$$\theta_{t+1} = \theta_t + \Delta\theta_t$$

$$\Delta\theta_t = -\lambda_t \frac{\sqrt{\delta_{t-1} + \epsilon}}{\sqrt{\mathbf{s}_t + \epsilon}} \nabla L(\theta)|_{\theta_t, X}$$

$$\delta_t = \beta\delta_{t-1} + (1 - \beta)(\Delta\theta_t)^2$$

Optimization Options – Momentum Methods

- ▶ *Intuition*: Keep going in the direction that was previously good, avoid "sharp turns"
- ▶ Standard momentum:

$$m_{t+1} = \beta m_t - \nabla L(\theta)|_{\theta_t, X}$$

$$\theta_{t+1} = \theta_t - \lambda m_{t+1}$$

Momentum

Parameter update

Typical β is ≈ 0.9

- ▶ Nesterov Momentum: Looks ahead and evaluates gradient at approximate next parameter values

$$m_{t+1} = \beta m_t - \lambda_t \nabla L(\theta)|_{\theta_t + \beta m_t, X}$$

$$\theta_{t+1} = \theta_t + m_{t+1}$$

Nesterov Momentum

Parameter update

Optimization Options – AdaM

Adaptive Moment Estimation

- Combine adaptive learning rate with momentum

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) \nabla L(\theta)|_{\theta_t, X}$$

$$s_t = \beta_2 s_{t-1} + (1 - \beta_2) (\nabla L(\theta)|_{\theta_t, X})^2$$

$$\theta_{t+1} = \theta_t - \lambda_t \frac{1}{\sqrt{s_t} + \epsilon} m_t$$

Training Neural Networks – Vanishing Gradients

Problem

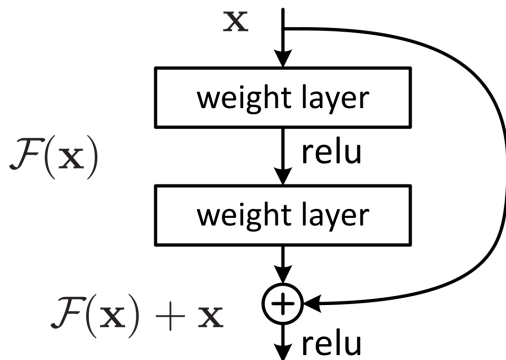
- ▶ Sigmoid and tanh functions are bounded for large positive or negative pre-activation values ("saturating activation functions")
- ▶ Long chains of neurons (e.g. in stacked fully-connected layers) can diminish the "error signal", i.e. the gradient

Possible Solutions

- ▶ Use non-saturating activation functions, e.g. ReLU, leaky ReLU, ELU, etc.
- ▶ Use additive rather multiplicative architectures, e.g. "ResNet" (residual networks)
- ▶ Standardize activations at every layer
- ▶ Carefully choose initial parameter values

Training Neural Networks – ResNet Architecture

- Allows gradients to bypass a layer that suffers from lack of learning (vanishing gradient, saturated activation)



Source: Murphy, Figure 13.15

Training Neural Networks – Exploding Gradients

Problem

- ▶ Long chains of neurons can vastly increase the error

Possible Solution

- ▶ Gradient clipping

$$g' = \min(1, \frac{1}{\|c\|_2})g$$

Training Neural Networks – Parameter Initialization Heuristics

- ▶ Random values from normal distribution: $\theta \sim N(0, \sigma^2)$
- ▶ "Xavier Initialization"/"Glorot Initialization"

$$\sigma^2 = \frac{2}{n_{\text{in}} + n_{\text{out}}}$$

where n_{in} is the number of incoming connections and n_{out} is the number of outgoing connections from each neuron

- ▶ "LeCun Initialization"

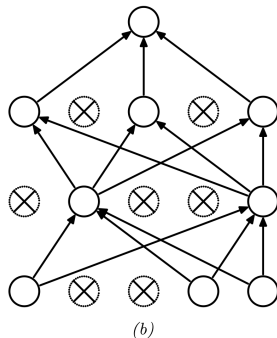
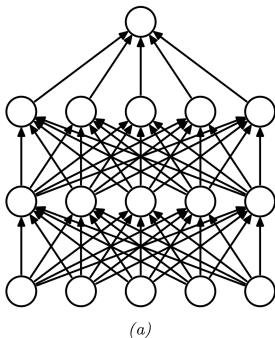
$$\sigma^2 = \frac{1}{n_{\text{in}}}$$

- ▶ "He Initialization"

$$\sigma^2 = \frac{2}{n_{\text{in}}}$$

Regularization – Dropout

- ▶ Randomly (per observation) remove a fraction of units in a layer, or equivalently,
- ▶ Randomly set the output of a fraction of units to 0
- ▶ Typically only done at train time, not test time



Source: Murphy, Figure 1.318



- ▶ Originally developed by Google, Version 1.0 in 2017
- ▶ Automatic differentiation/gradients
- ▶ Distributed computing
- ▶ Parallel computing on multiple GPU
- ▶ Wide range of loss functions
- ▶ Wide range of optimizers
- ▶ Wide range of neural network types and activation functions



- ▶ Originally developed as a user-friendly, high-level abstraction layer for different ML frameworks, including Tensorflow, Theano, PyTorch
- ▶ Wide range of standard neural network layers
- ▶ Simplified training loops

Regression using Keras

Import required packages:

```
import pandas as pd
import numpy as np
import tensorflow as tf
from tensorflow.keras import layers
```

Read a CSV file:

```
# Use the Boston housing data set
boston_data = \
    pd.read_csv("https://evermann.ca/bus14720/boston.csv")
```

Separate features and target:

```
boston_features = boston_data.copy()
boston_labels = boston_features.pop('medv')
```

Regression using Keras [cont'd]

Define the NN model with one hidden fully-connected layer (64 neurons) and one fully-connected output layer (1 neuron) in sequence. No activation function is given, so this is a linear regression model:

```
boston_model = tf.keras.Sequential([
    layers.Dense(64, activation=None),
    layers.Dense(1, activation=None)
])
```

Set the loss function and the optimizer:

```
boston_model.compile(
    loss = tf.keras.losses.MeanSquaredError(),
    optimizer = tf.keras.optimizers.Adam())
```

Fit/train the model for 25 epochs:

```
boston_model.fit(boston_features,
                  boston_labels, epochs=25)
boston_model.summary()
```

Regression in Keras [cont'd]

The `Normalization` layer normalizes numeric features:

```
norm_layer = layers.Normalization()
```

The `adapt()` function computes means and variances of the data so the layer can normalize the data when the model is fit. Requires numpy array.

```
norm_layer.adapt(boston_features.to_numpy())
```

Add the normalization layer to the model. A ReLU activation is used that makes this a non-linear regression model:

```
norm_boston_model = tf.keras.Sequential([
    norm_layer,
    layers.Dense(64, activation='relu'),
    layers.Dense(1, activation=None)
])
```

Regression in Keras [cont'd]

Set loss and optimizer and ask Keras to keep track of the MSE and MAE metrics.

```
norm_boston_model.compile(  
    loss = tf.keras.losses.MeanSquaredError(),  
    optimizer = tf.keras.optimizers.Adam(),  
    metrics = ['mse', 'mae'])
```

The `fit` function returns a history of the metrics we asked for:

```
train_hist = \  
    norm_boston_model.fit(  
        boston_features,  
        boston_labels,  
        batch_size=20,  
        epochs=50,  
        validation_split=0.33)
```


Plot the training history using Plotly Express

```
import plotly.express as px

hist = pd.DataFrame({
    'training': train_hist.history['mse'],
    'validation': train_hist.history['val_mse']})
hist['epoch'] = np.arange(hist.shape[0])
hist = pd.melt(hist,
               id_vars='epoch',
               value_vars=['training', 'validation'])

fig = px.line(hist, x='epoch', y='value',
              color='variable')
fig.show()
```

- ▶ Modify the above code to include different activation functions, e.g. `"tanh"`, `"sigmoid"`, or `"elu"`. Comment on the learning progress and loss function values.
- ▶ Modify the above code to change the number of neurons in the `"Dense"` layer. Comment on the learning progress and loss function values.
- ▶ Modify the architecture to add one or more `"Dense"` layers with different numbers of units. Comment on the learning progress and loss function values.

The `Wage` dataset from the `ISLR2` library for R has been adapted to include a column `wagequart`, the quartile of the wage. Many features are categorical.

```
# Read data and separate features from target labels
wage_data = \
    pd.read_csv("https://evermann.ca/wage.csv")

wage_features = wage_data.copy()
wage_labels = wage_features.pop('wagequart') - 1
```

Treat each categorical string feature and convert to **one-hot encoding**.

One-hot encoding is similar to binary dummy variables (contrasts) in linear models, but have no default level; a feature with n different categories requires n binary variables (not $n - 1$ as in linear model contrasts).

Keep track of the inputs and the pre-processed inputs:

```
inputs = {}  
preproc_inputs = []
```

Classification in Keras [cont'd]

```
for cat_feature in ['maritl', 'race', 'education', \
                    'jobclass', 'health', 'health_ins']:
    # An Input variable is a placeholder that
    # accepts data input when training or predicting
    input = tf.keras.Input(shape=(1,),
                            name=cat_feature,
                            dtype=tf.string)

    # This StringLookup layer accepts a string and
    # outputs its category as a one-hot vector
    lookup = layers.StringLookup(
        name=cat_feature+"_lookup",
        output_mode="one_hot")

    # Adapt it to the different strings in the data
    lookup.adapt(wage_features[cat_feature])
    # And tie the input to this layer
    onehot = lookup(input)

    inputs[cat_feature] = input
    preproc_inputs.append(onehot)
```

Define and input and a Normalization layer for the numerical variable `age`:

```
age_input = tf.keras.Input(shape=(1,),
                             name="age",
                             dtype="float32")
norm_layer = layers.Normalization(name="age_norm")
norm_layer.adapt(wage_features["age"])
age_norm = norm_layer(age_input)

inputs["age"] = age_input
preproc_inputs.append(age_norm)
```

Define and input and a one-hot encoding `IntegerLookup` layer for the numeric variable `year`:

```
year_input = tf.keras.Input(shape=(1,),
                             name="year",
                             dtype="int32")
lookup = layers.IntegerLookup(name="year_lookup",
                              output_mode="one_hot")
lookup.adapt(wage_features["year"])
year_onehot = lookup(year_input)

inputs["year"] = year_input
preproc_inputs.append(year_onehot)
```

Concatenate the pre-processing outputs to one long vector with a `Concatenate` layer. Call this layer with the list of pre-processed inputs:

```
preprocessed_inputs = \
    layers.Concatenate(name="concat")(preproc_inputs)
```

Build a pre-processing model whose inputs is the dict of `Input` variables, and whose output are the results of calling the layers:

```
preproc_model = tf.keras.Model(inputs,
                                preprocessed_inputs,
                                name="preproc")
preproc_model.summary()
```


Build the classification model as a Sequential model:

```
class_model = tf.keras.Sequential(name="classification")
class_model.add(layers.Dense(64, activation="relu"))
class_model.add(layers.Dropout(0.25))
class_model.add(layers.Dense(32, activation="relu"))
class_model.add(layers.Dropout(0.25))
class_model.add(layers.Dense(4, activation="softmax"))

# Alternatively:
# class_model.add(layers.Dense(4, activation=None))
# class_model.add(layers.Softmax())
```

The output of the pre-processing model is the input to the classification model:

```
class_results = class_model(preproc_model(inputs))  
class_model.summary()
```

The final model takes the inputs, and has the classification model results as outputs:

```
wage_model = tf.keras.Model(inputs, class_results,  
                             name="wage_model")  
wage_model.summary()
```

Compile the model with loss function, optimizer and request training metrics:

```
wage_model.compile(  
    loss=tf.keras.losses.SparseCategoricalCrossentropy(  
        from_logits=False),  
    optimizer=tf.keras.optimizers.Adam(  
        learning_rate=0.001,  
        beta_1 = 0.9,  
        beta_2 = 0.999,  
        epsilon = 1e-07),  
    metrics=[  
        tf.keras.metrics.SparseCategoricalAccuracy(),  
        tf.keras.metrics.KLDivergence()])
```

Note: Specifying `from_logits=True` for the loss can save the softmax activation or layer at the bottom of the sequential classification model.

Create the input data as a dict of numpy arrays to match the Input variables:

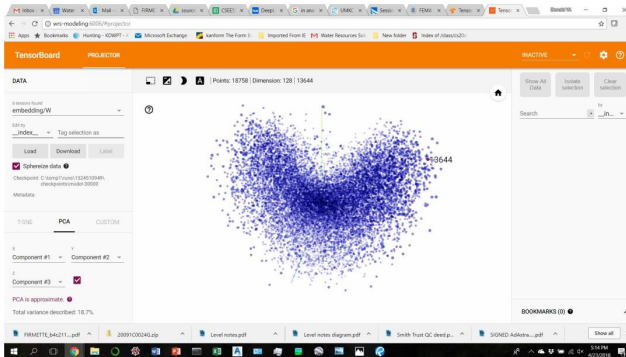
```
import numpy as np
wage_feature_dict = \
    {name: np.array(value) for \
     name, value in wage_features.items() }
```

Write log information to a directory for loading into **Tensorboard**:

```
import datetime
log_dir = "./tensorboard_logs/" + \
    datetime.datetime.now().strftime("%Y%m%d-%H%M%S")
tensorboard_callback = \
    tf.keras.callbacks.TensorBoard(log_dir=log_dir,
                                   histogram_freq=0)
```

TensorBoard

TensorBoard is a tool to visualize neural network models and their training and validation data/performance.



https://commons.wikimedia.org/wiki/File:Tensorboard_1.jpg

Train the model for 25 epochs:

```
wage_hist = wage_model.fit(  
    x = wage_feature_dict,  
    y = wage_labels,  
    validation_split=0.333,  
    batch_size=20,  
    epochs = 25,  
    callbacks=[tensorboard_callback])
```

Plot the training history using Plotly Express

```
import plotly.express as px

hist = pd.DataFrame({
    'training': \
wage_hist.history['sparse_categorical_accuracy'],
    'validation': \
wage_hist.history['val_sparse_categorical_accuracy']})
hist['epoch'] = np.arange(hist.shape[0])
hist = pd.melt(hist,
               id_vars='epoch',
               value_vars=['training', 'validation'])

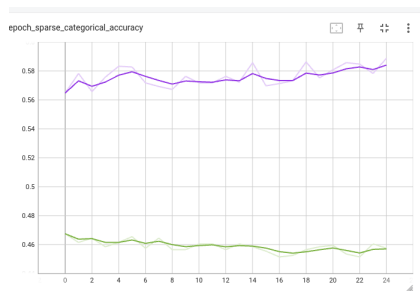
fig = px.line(hist, x='epoch', y='value',
              color='variable')
fig.show()
```

Classification in Keras [cont'd]

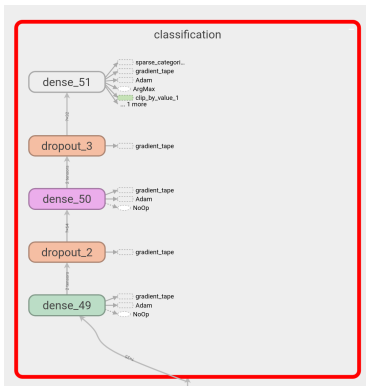
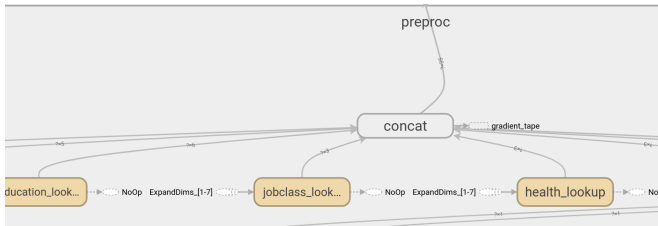
Call TensorBoard from the terminal, providing the log directory

```
tensorboard --logdir tensorboard_logs
```

Then go to `http://localhost:6006` in your web browser.



TensorBoard



Another useful callback function:

```
earlystop_callback = tf.keras.callbacks.EarlyStopping(  
    monitor = 'val_loss',  
    patience = 3,  
    mode = 'min',  
    # or 'max' or 'auto' depending on monitor metric  
    restore_best_weights = True)
```

Hands-On Exercises

- ▶ Examine the model summaries for the pre-processing, the classification, and the complete wage model. Explain the number of trainable and total parameters, and also explain the output shapes of each layer.
- ▶ Make the "wage" prediction a binary classification problem:
 - 1 Modify the `wage_labels` and combine classes 0, 1 and classes 2, 3 (class numbers should be 0 or 1)
 - 2 Modify the classification network to have a single output node
 - 3 Use the `BinaryCrossentropy` loss
 - 4 Return the following metrics as part of the training history:
 - ▶ Precision
 - ▶ Recall
 - ▶ AUC
 - 5 Plot the metrics after training