U O W

Introduction to ANN and TensorFlow/Keras

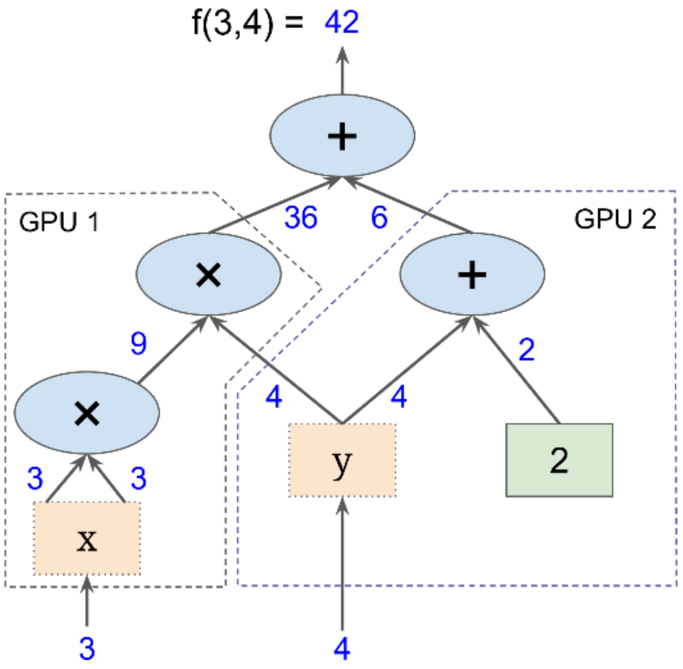
CSCI316: Big Data Mining Techniques and Implementation

UNIVERSITY
OF WOLLONGONG
AUSTRALIA

Contents

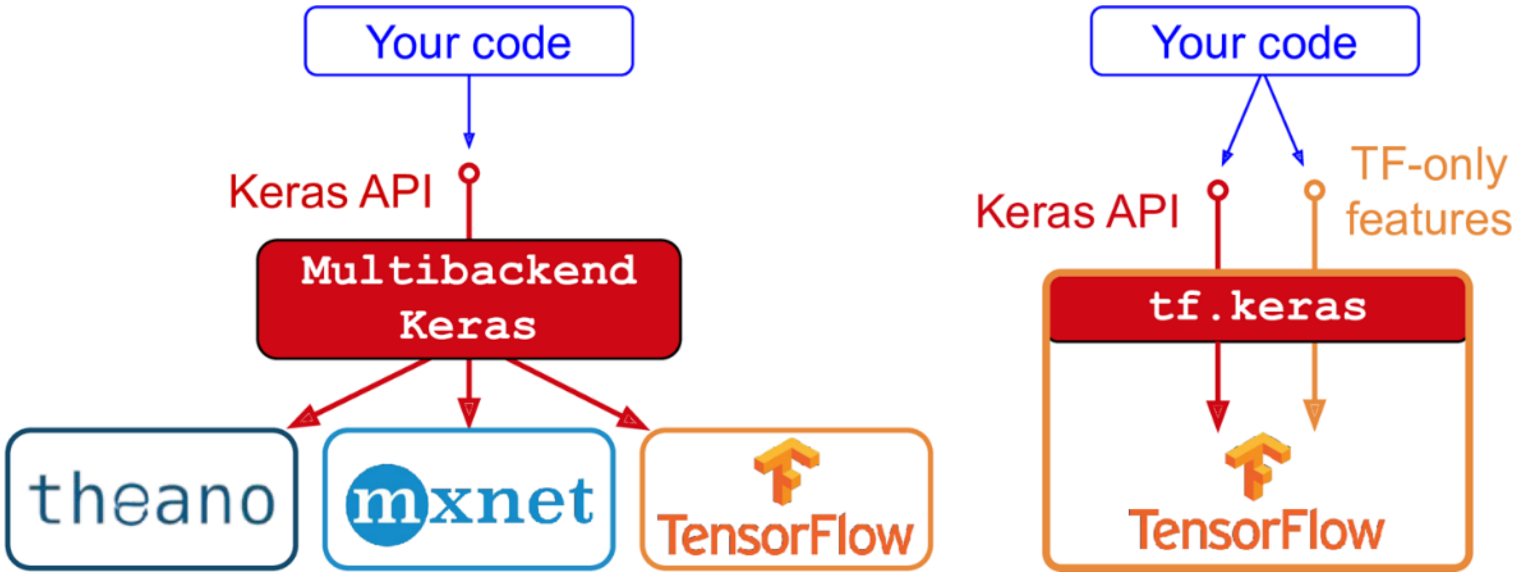
What is TensorFlow?

- TensorFlow is a Python-friendly open source library for numerical computation well-suited for large-scale ML and deep learning.
- Some key features:
  - In TensorFlow, define in Python a graph of computations to perform.
  - TensorFlow breaks the graph into chunks and run them in parallel across multiple CPU, GPU and TPU.



UNIVERSITY OF WOLLONGONG AUSTRALIA

TensorFlow and Keras

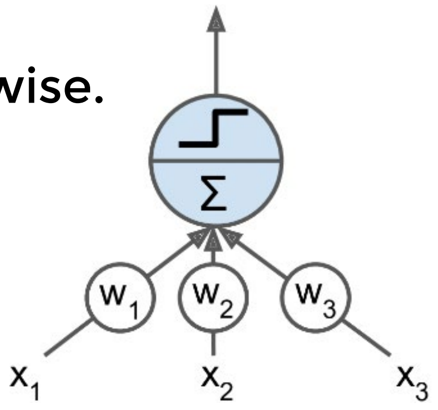


The Linear Threshold Unit (LTU)

- Inputs of a LTU are numbers
- Each input connection is associated with a weight.
- Computes a weighted sum of its inputs and applies a step function to that sum.

- $z = w_1x_1 + w_2x_2 \ldots + w_nx_n = \boldsymbol{w}^T\boldsymbol{x}$
- $\hat{y} = step(z) = step(\boldsymbol{w}^T\boldsymbol{x})$

Note. $step(z)$ is 0 if $z < 0$ and 1 otherwise.

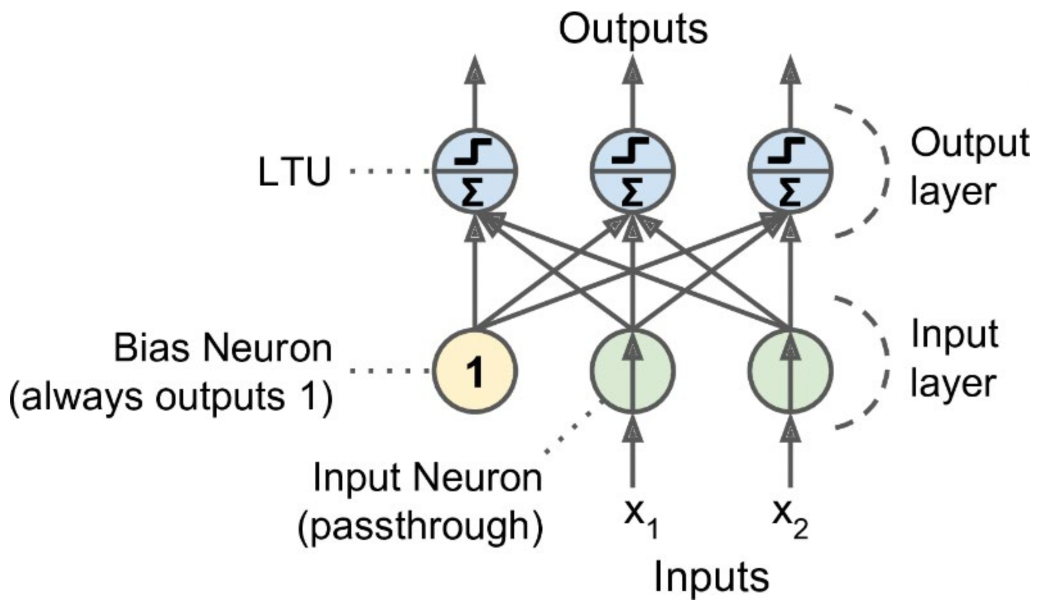


The Perceptron

- The perceptron is a single layer of LTUs.
- The input neurons output whatever input they are fed.
- A bias neuron, which just outputs 1 all the time.
- If we use the **logistic function (sigmoid)** instead of a step function, it computes a continuous output.

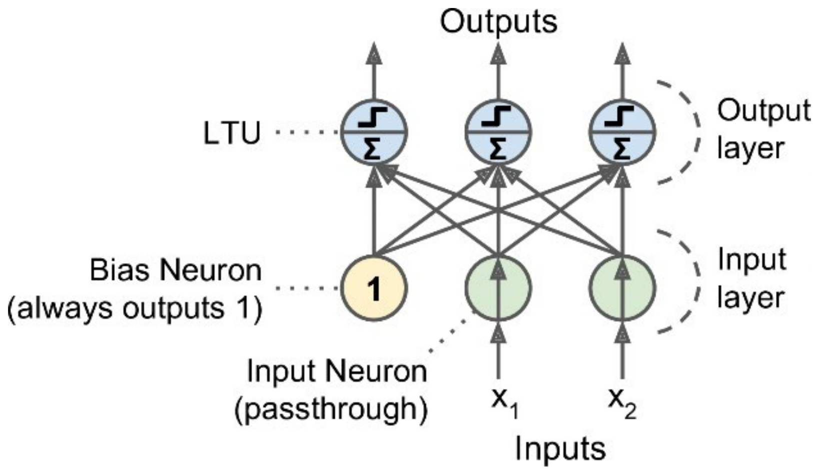
Sigmoid function:

$$\sigma(x) = \frac{e^x}{1 + e^x}$$



How is a Perceptron Trained

- Feed a training instance $\boldsymbol{x}$ to each output neuron j at a time and makes it prediction $\hat{y}_j$.
- Updates the connection weights (***gradient decent***)
  - $\hat{y}_j = \sigma(\boldsymbol{w}_j^T \boldsymbol{x} + b_j)$
  - $J(\boldsymbol{w}_j, b_j) = mse(y_j, \hat{y}_j)$
  - $w_{i,j}^{(new)} = w_{i,j} - \eta \frac{\partial J(\boldsymbol{w}_j, b_j)}{\partial w_{i,j}}$
    - $w_{i,j}$: the weight between neuron i & j
    - x_i: the i-th input value
    - $\hat{y}_j$: the j-th predicted output value
    - y_j: the j-th true output value
    - η: the learning rate (i.e. step size)
  - $b_j^{(new)} = b_j - \eta \frac{\partial J(\boldsymbol{w}_j, b_j)}{\partial b_j}$



Note. $\sigma(\boldsymbol{w}_j^T \boldsymbol{x} + b_j)$ is a logistic regression classifier when it is applied to a binary classification problem. $\hat{y}_j$ is a probabilistic prediction

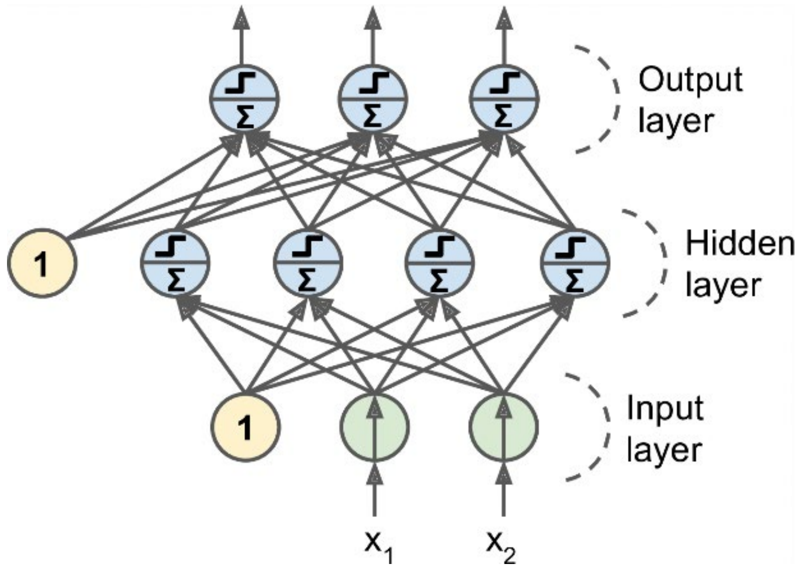
Matrix-Vector Notations

Use the Matrix-Vector notations to represent the math equations in a more compact way:

- $\hat{y}_j = \sigma\left(\boldsymbol{w}_j^T \boldsymbol{x} + b_j\right) \implies \hat{\boldsymbol{y}} = \sigma(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b})$
- $J\left(\boldsymbol{w}_j, b_j\right) = mse\left(y_j, \hat{y}_i\right) \implies J(\boldsymbol{W}, \boldsymbol{b}) = mse(\boldsymbol{y}, \hat{\boldsymbol{y}})$
  - (if MSE is used)
- $w_{i,j}^{(new)} = w_{i,j} - \eta \frac{\partial J\left(\boldsymbol{w}_j, b_j\right)}{\partial w_{i,j}} \implies \boldsymbol{W}^{(new)} = \boldsymbol{W} - \eta \frac{\partial J(\boldsymbol{W}, \boldsymbol{b})}{\partial \boldsymbol{W}}$
- $b_j^{(new)} = b_j - \eta \frac{\partial J\left(\boldsymbol{w}_j, b_j\right)}{\partial b_j} \implies \boldsymbol{b}^{(new)} = \boldsymbol{b} - \eta \frac{\partial J(\boldsymbol{W}, \boldsymbol{b})}{\partial \boldsymbol{b}}$

Multi-Layer Perceptron (MLP)

- Stacking multiple Perceptron into a network can dramatically improved the expressive power.
- The resulting network is called an MLP or **feedforward neural network**, which is the basic model for other kinds of ANNs.
- A feedforward neural network is composed of
  - One input layer
  - One or more hidden layers
  - One final output layer
- Every layer except the output layer includes a bias neuron and is fully connected to the next layer.
- In theory, a sufficiently large MLP can approximate any continuous function



UNIVERSITY OF WOLLONGONG AUSTRALIA

MLP– Cost Function

- For **a regression problem**, we use the MSE as the ***cost function*** (a.k.a. ***loss function***) – see the previous lecture
- For a (**multi-)classification problem**, we usually **cross entropy** (i.e., *negative log-likelihood*) between the true class labels $\boldsymbol{y}$ of data and the model's class predictions $\hat{\boldsymbol{y}}$
  - Let $\hat{y}_i^{(j)}$ denote the predicted *probability* that record i belongs to class j. And let $y_i^{(j)}$ is 1 if j is the true class label of record i, and $y_i^{(j)}$ is 0 otherwise.
  - Then, $\text{crossentropy}(y_i, \hat{y}_i) := -\sum_j y_i^{(j)} \log(\hat{y}_i^{(j)})$

  (compare it with Shannon's entropy function in DT)
  - The ***cross-entropy loss*** is defined as:

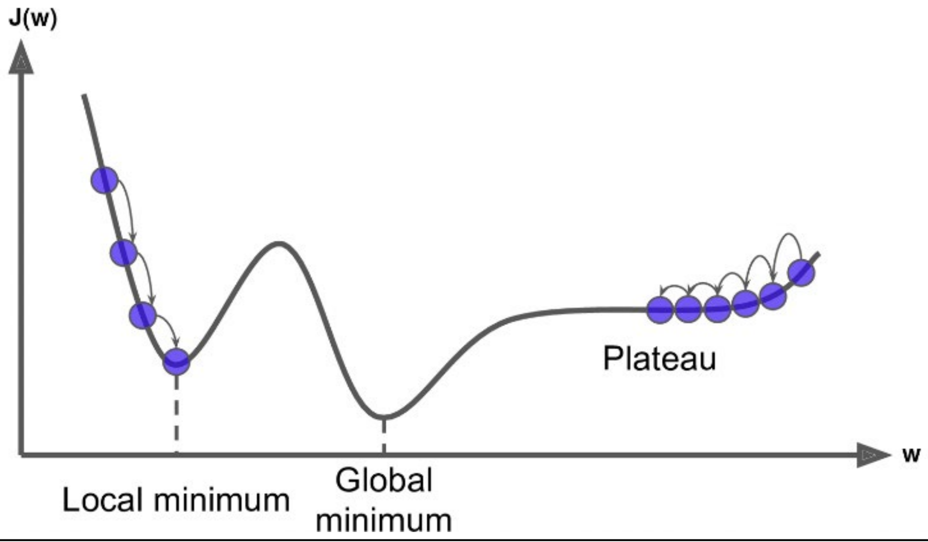
$$\text{cost}(\boldsymbol{y}, \hat{\boldsymbol{y}}) := \frac{1}{m}\sum_{i=1}^{m} \text{crossentropy}(y_i, \hat{y}_i)$$

  where m is the size of $\boldsymbol{y}$ and $\hat{\boldsymbol{y}}$ (i.e., the total number of records).

Gradient-Based Learning

- The most significant difference between the linear models (e.g., linear regression) and feedforward neural network?
- Linear models, with convex cost functions, guarantee to find global minimum.

  *Convex optimisation converges starting from any initial parameters.
- However, the non-linearity of a neural network causes its cost functions to be non-convex.



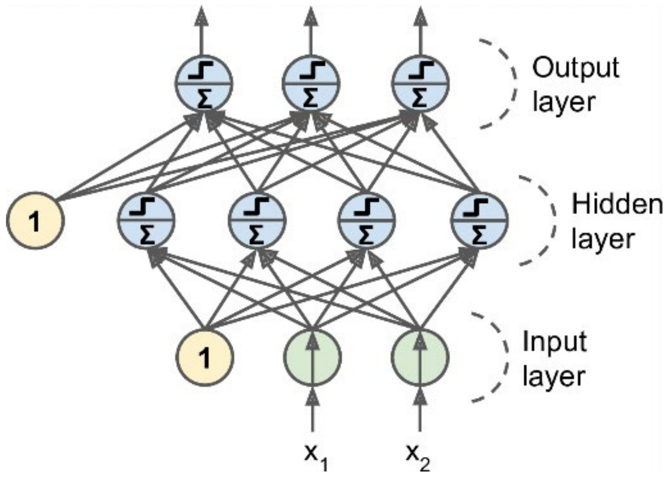
Gradient-Based Learning

- **Gradient descent** applied to non-convex cost functions has no such convergence guarantee.
- It is sensitive to the values of initial parameters.
- For feedforward neural networks, it is important to initialise all *weights* to small random values.
- The *biases* may be initialised to zero or to small positive values.
- Improvement: **Stochastic or minibatch gradient descent**
  - Helps to “jump out” of local minima
  - Reasonable number of epochs until convergence
  - Can benefit from parallel computation

UNIVERSITY
OF WOLLONGONG
AUSTRALIA

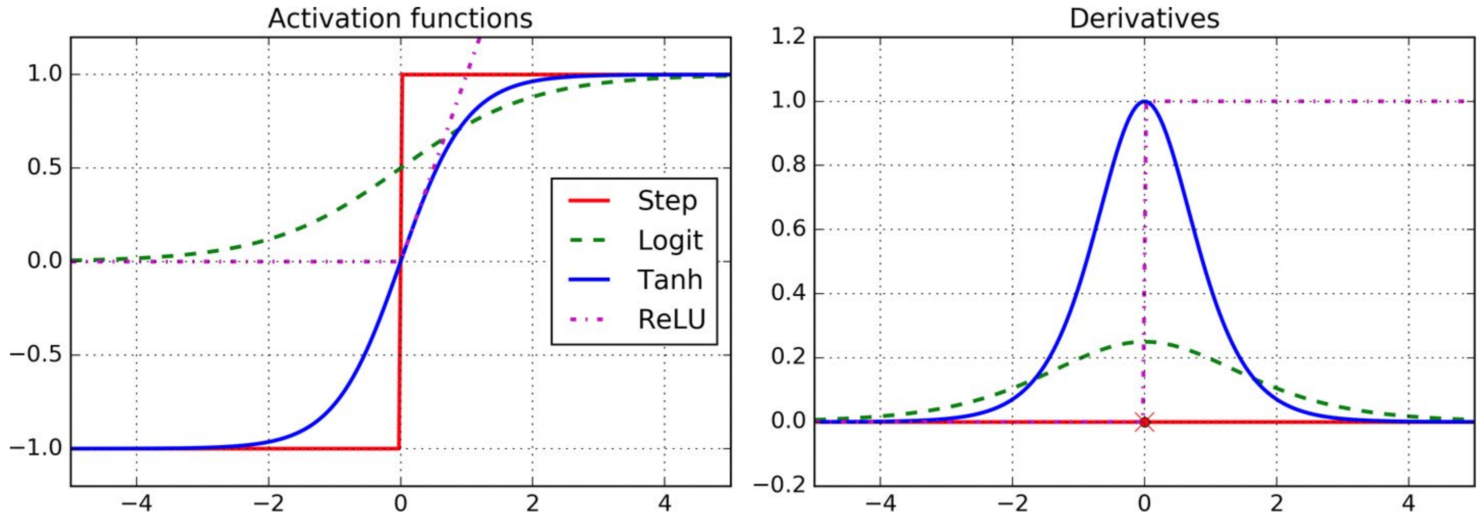
Training Feedforward Neural Network

- How to train a feedforward neural network?
  - ANN training is challenging.
- For each training instance $\boldsymbol{x}^{(i)}$ the algorithm does the following:
  - Forward pass: make a prediction (compute $\hat{y}^{(i)} = f(\boldsymbol{x}^{(i)})$).
  - Measure the error (compute $cost(y, \hat{y})$).
  - Backward pass: go through each layer *in reverse* to measure the error contribution from each connection
  - Tweak the connection weights and biases to reduce the error (update $\boldsymbol{W}$ and $\boldsymbol{b}$).
- It is called the **backpropagation** training algorithm.
  - More in later slides…



Hidden Units

- In order for the training algorithm for MLP to work properly, we need to replace the step function with other activation functions (why?)
- Alternative activation functions:
  - Logistic function (sigmoid): $\sigma(z) = \frac{e^x}{1+e^x} = \frac{1}{1+e^{-z}}$
  - Hyperbolic tangent function: $\tanh(z) = 2\,\sigma(2z) - 1$
  - Rectified linear units (ReLUs): $\mathrm{ReLU}(z) = \max(0, z)$

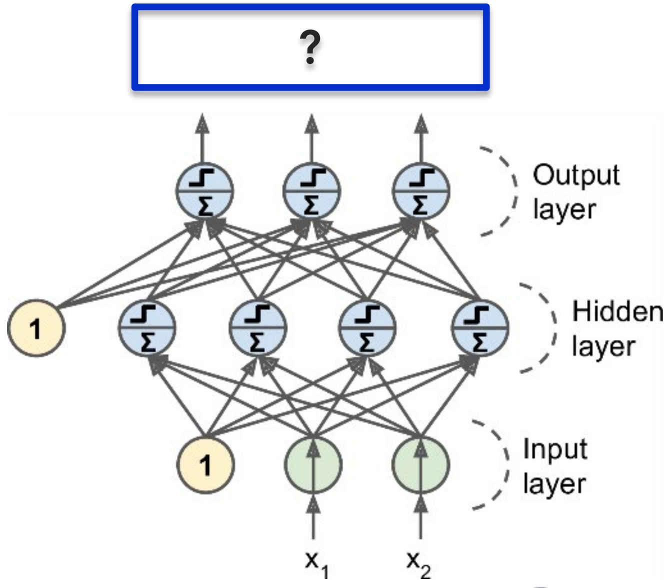


UNIVERSITY OF WOLLONGONG AUSTRALIA

Output Units

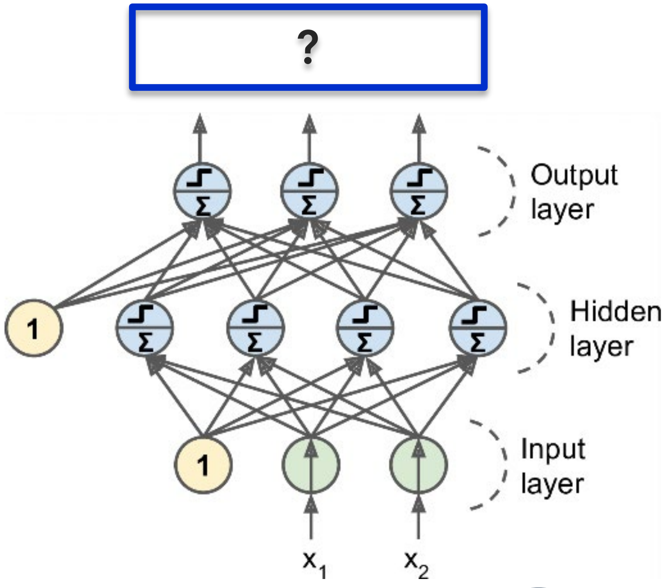
- MLP includes three types of layers: **input**, **hidden** and **output** layers.
- Usually, the output layer in MLP includes **linear** units, **sigmoid** units (for binary classification) or **softmax** units (for multinomial classification).
- Given ***h*** as the output neurons in the layer before the output layer.

- In the *first two* cases, each neuron *j* in the output layer produces
  - $\hat{y}_j = \boldsymbol{w}_j^T \boldsymbol{h} + b_j$ produces the input of output neuro j (i.e., applying *none*), or
  - $\hat{y}_j = \sigma(\boldsymbol{w}_j^T \boldsymbol{h} + b_j)$ applying sigmoid to output neuro j
- Minimising the MSE or cross-entropy



Output Units

- In the *last* case, each neuron j in the output layer produces
  - All inputs: $\boldsymbol{z} = \boldsymbol{W}\boldsymbol{h} + \boldsymbol{b}$
  - Define the *softmax* function:
    $$\text{softmax}_j(\boldsymbol{z}) = \frac{e^{z_j}}{\sum_{i=1}^{|z|} e^{z_i}}$$
  - Then, $\hat{y}_j = \text{softmax}_j(\boldsymbol{z})$, i.e., applying softmax to the inputs
- Minimising the cross-entropy



Typical Network Architecture for Regression

Hyperparameters	Typical Values
# input neurons	One per input feature (e.g., 28 x 28 = 784 for MNIST)
# hidden layers	Depends on the problem. Typically 1 to 20.
# neurons per hidden layer	Depends on the problem. Typical 10 to 100 for small networks but it can be very large.
# output neurons	1 per prediction dimension
Hidden activation	ReLU, Logistic or Tanh
Output value	None or ReLU (if positive outputs) or Logistic/Tanh (if bounded outputs)
Loss function	MSE (or others)

Typical Network Architecture for Classification

Hyperparameter	Binary classification	Multi-label binary classification	Multi-class classification
Input and hidden layers	Same as regression	Same as regression	Same as regression
# output neurons	1	1 per label	1 per class
Output layer activation	Logistic	Logistic	Softmax
Loss function	Cross Entropy	Cross Entropy	Cross Entropy

Fashion MNIST Data Set

> import tensorflow as tf
> from tensorflow import keras

- Example data set: **Fashion MNIST**

> fashion_mnist = keras.datasets.fashion_mnist (X_train_full, y_train_full)
> (X_test, y_test) = fashion_mnist.load_data()

Implement MLP in Keras for Classification

- Check the same and data type:

```
> X_train_full.shape        # (60000, 28, 28)
  X_train_full.dtype        # dtype('uint8')
```

- Scale the pixel intensities down to the [0, 1] range by dividing them by 255.0 (this also converts them to floats):

```
> X_valid, X_train = X_train_full[:5000] / 255.0, X_train_full[5000:] / 255.0
  y_valid, y_train = y_train_full[:5000], y_train_full[5000:]
```

- The list of class names:

```
> class_names = ["T-shirt/top", "Trouser", "Pullover", "Dress", "Coat",
  "Sandal", "Shirt", "Sneaker", "Bag", "Ankle boot"]
> class_names[y_train[0]]          # 'Coat'
```

Implement MLP in Keras for Classification

Creating an MLP using the Sequential API

```
> model = keras.models.Sequential()
  model.add(keras.layers.Flatten(input_shape=[28, 28]))
  model.add(keras.layers.Dense(300, activation="relu"))
  model.add(keras.layers.Dense(100, activation="relu"))
  model.add(keras.layers.Dense(10, activation="softmax"))
```

- The first layer flattens the input from 2D to 1D (why the data is 2D?)
- Specifying *activation="relu"* is equivalent to specifying *activation=keras.activations.relu.**

*More about activation functions implemented in Keras in later slides.

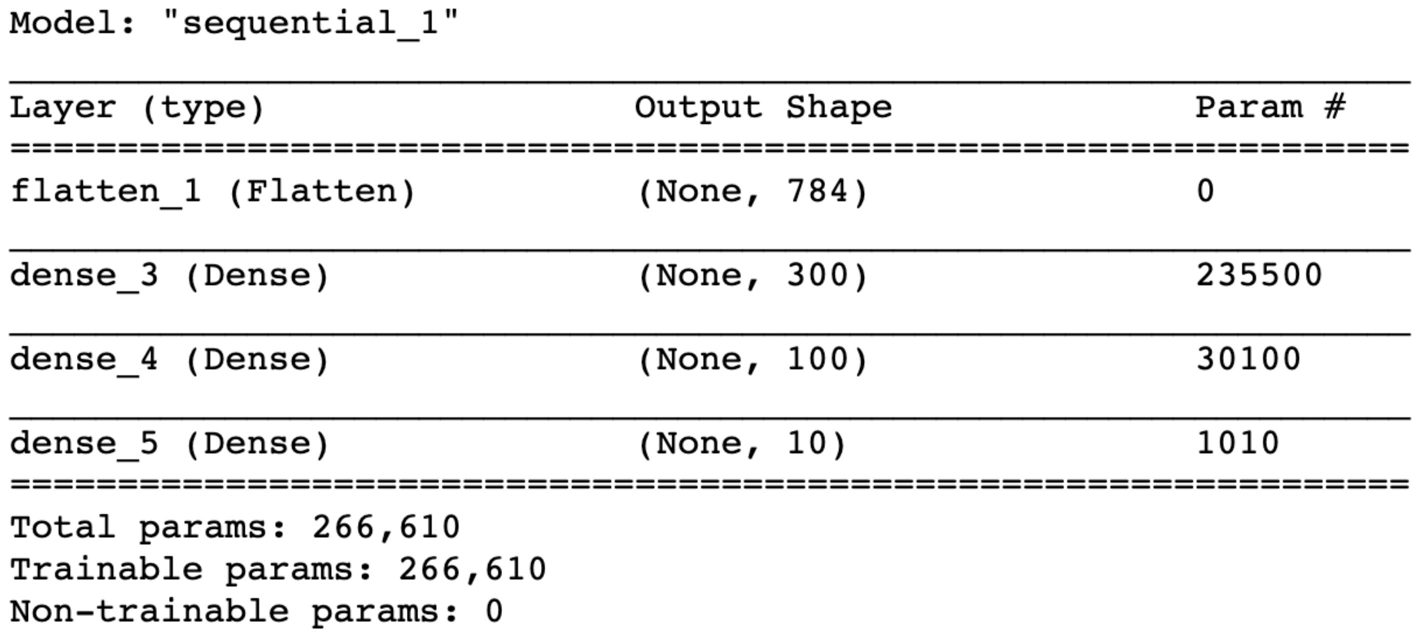
- Alternatively:

```
> model = keras.models.Sequential([
   keras.layers.Flatten(input_shape=[28, 28]),
   keras.layers.Dense(300, activation="relu"),
   keras.layers.Dense(100, activation="relu"),
   keras.layers.Dense(10, activation="softmax") ])
```

Implement MLP in Keras

- Get a list of model layers and trainable model parameters

> model.summary()



```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten_1 (Flatten)          (None, 784)               0
_________________________________________________________________
dense_3 (Dense)              (None, 300)               235500
_________________________________________________________________
dense_4 (Dense)              (None, 100)               30100
_________________________________________________________________
dense_5 (Dense)              (None, 10)                1010
=================================================================
Total params: 266,610
Trainable params: 266,610
Non-trainable params: 0
```

Implement MLP in Keras

- Can check the parameter values of each layer. For example:

```
> hidden1 = model.layers[1]
> weights, biases = hidden1.get_weights()
  # array([[ 0.02448617, -0.00877795, -0.02189048, ..., -0.02766046,
  0.03859074, -0.06889391], ..., [-0.06022581, 0.01577859, -0.02585464,
  ..., -0.00527829, 0.00272203, -0.06793761]], dtype=float32)
> weights.shape  # (784, 300)
> biases # array([0., 0., 0., 0., 0., 0., 0., 0., 0., ..., 0., 0., 0.], dtype=float32)
> biases.shape # (300,)
```

- Dense layer initialized the connection weights randomly, and the biases were initialized to zeros, which is fine.

Implement MLP in Keras

Compiling the model

- After a model is created, call its *compiling* method to specify the loss function and the optimizer to use.

```
> model.compile(loss="sparse_categorical_crossentropy",
  optimizer="SGD", metrics=["accuracy"])
```

- Some parameters are set: **loss**, **optimizer** and **metrics**.

Implement MLP in Keras

Compiling the model

> model.compile(**loss**="sparse_categorical_crossentropy", optimizer="sgd", metrics=["accuracy"])

- The "sparse_categorical_crossentropy" loss is used as we have sparse labels (i.e., for each instance, there is just one target class index, from 0 to 9), and the classes are *exclusive*.
- Note. If we had one target probability per class for each instance (such as one-hot vectors, e.g. [0., 0., 0., 1., 0., 0., 0., 0., 0., 0.] to represent class 3), then use the "categorical_crossentropy" loss instead.*

*More about loss functions implemented in Keras in later slides.

- Note. If we were doing binary classification (with one or more binary labels), then we would use the "sigmoid" (i.e., logistic) activation function in the output layer instead of the "softmax", and we would use the "binary_crossentropy" loss.

❑ Official docs: https://keras.io/metrics

Implement MLP in Keras

Compiling the model

> model.compile(loss="sparse_categorical_crossentropy", optimizer="SGD", metrics=["accuracy"])

- “SGD” means that we will train the model using simple *Stochastic Gradient Descent*
  - ❖ Maybe minibatch with a default batch size; in this case, if real SGD has to be used, set the batch size to 0.
- Since this MLP is a classifier, we use “ ” as the performance metric for training and evaluation. (Can specify multiple metrics.)*

*More about optimizers implemented in Keras in later slides.

Implement MLP in Keras

Training and evaluating the model

- Train the model by calling its *fitting* method:

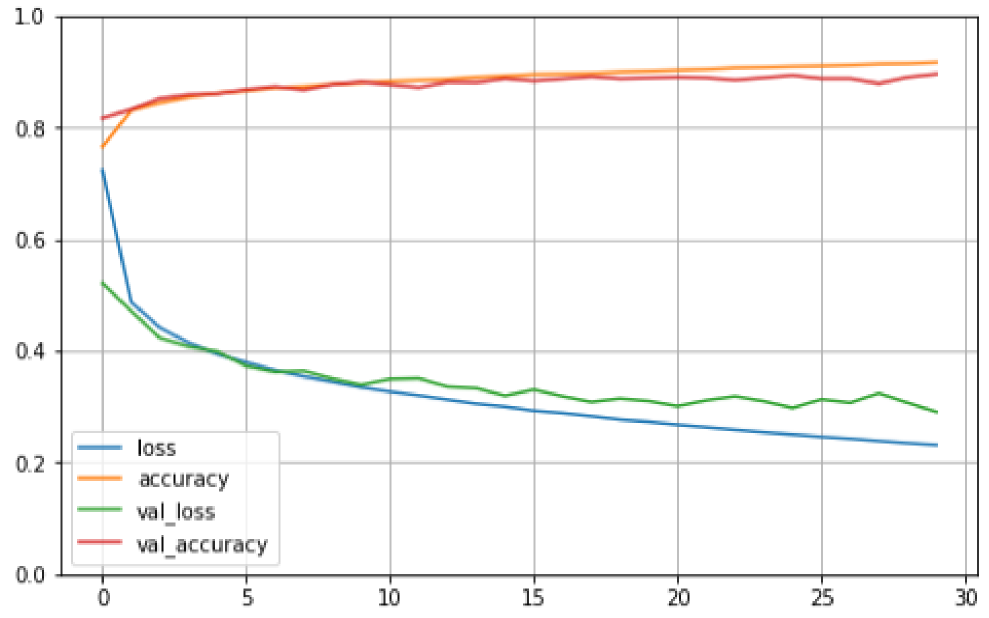
```
> history = model.fit(X_train, y_train, epochs=30, validation_data=(X_valid, y_valid))
```

- During this process, the weights and bias of each layer in the model are tweaked using the training data set, and the accuracy are evaluated using the validation data set.
- fit() returns a history object contains the progressive information of the process (called ***learning curve***).

Implement MLP in Keras

- Plot the learning curve:

```
> import pandas as pd
  import matplotlib.pyplot as plt
  pd.DataFrame(history.history).plot(figsize=(8, 5))
  plt.grid(True)
  plt.gca().set_ylim(0, 1) # set the vertical range to [0-1]
  plt.show()
```



UNIVERSITY OF WOLLONGONG AUSTRALIA

Implement MLP in Keras

Test the model and make prediction:

```
> model.evaluate(X_test, y_test)
  10000/10000 [==========] - 0s 29us/sample - loss: 0.3340 - accuracy:
  0.8851 [0.3339798209667206, 0.8851]
```

```
> X_new = X_test[:3]
  y_proba = model.predict(X_new)
  y_proba.round(2)
  # array([[0., 0., 0., 0., 0., 0., 0., 0., 0., 1.], [0., 0., 1., 0., 0., 0., 0., 0., 0., 0.],
  [0., 1., 0., 0., 0., 0., 0., 0., 0., 0.]], dtype=float32)
```

```
> import numpy as np
  y_pred = np.argmax(y_proba, axis=-1)
  print(y_pred) # [9,2,1]
```

UNIVERSITY OF WOLLONGONG AUSTRALIA

Fine-Tuning Neural Network Hyperparameters

- We use the **Grid Search (with Cross Validation)** in Scikit-Learn to search the best ***hyperparameters*** in a grid.
  - To this end, we *wrap* our Keras models in objects that mimic regular Scikit-Learn classifier.

```
> def build_model(n_hidden=1, n_neurons=30, input_shape=[28,28]):
      model = keras.models.Sequential()
      model.add(keras.layers.Flatten(input_shape=input_shape))
      for layer in range(n_hidden):
          model.add(keras.layers.Dense(n_neurons, activation="relu"))
      model.add(keras.layers.Dense(10, activation="softmax"))
      model.compile(optimizer="SGD",
          loss="sparse_categorical_crossentropy", metrics=["accuracy"])
      return model
> keras_clf = keras.wrappers.scikit_learn.KerasClassifier(build_model)
```

❖ Note. If it is a regression problem, use a “regressor wrapper”: **keras.wrappers.scikit_learn.KerasRegressor**().

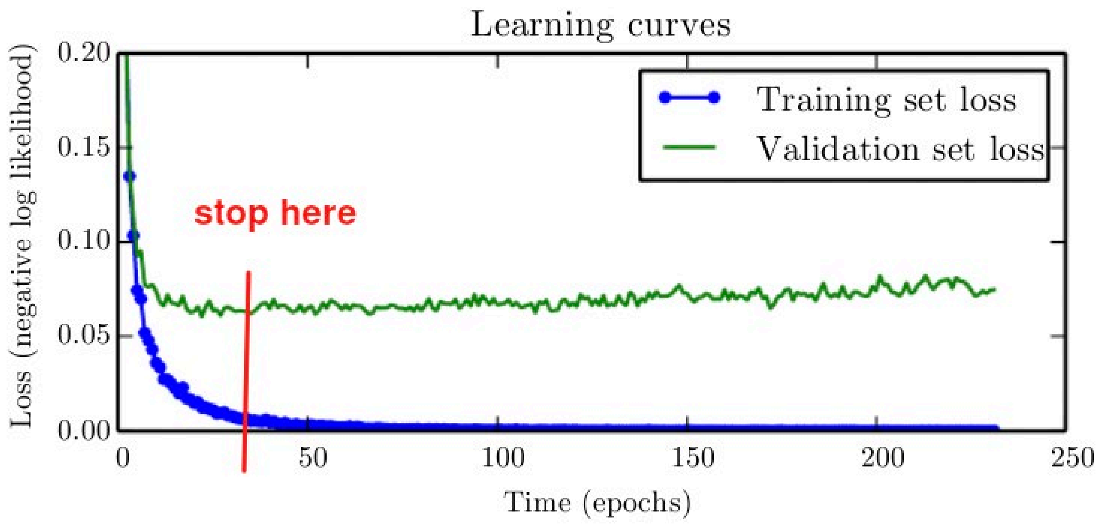
UNIVERSITY OF WOLLONGONG AUSTRALIA

Fine-Tuning Neural Network Hyperparameters

```
>   from sklearn.model_selection import GridSearchCV
    param_distribs = {
            "n_hidden": [1, 2],
            "n_neurons": [50,100]
        }
>   rnd_search_cv = GridSearchCV(keras_clf, param_distribs, cv=5)
    rnd_search_cv.fit(X_train, y_train, epochs=10,
        validation_data=(X_valid, y_valid),
        callbacks=[keras.callbacks.EarlyStopping(patience=10)]) # see next slide
>   rnd_search_cv.best_params_ # {'n_hidden': 2, 'n_neurons': 100}
>   rnd_search_cv.best_score_ # 0.8909999895095826
```

Early stopping

- As the training steps go by, its prediction error on the training/validation set naturally goes down.
- After a while the validation error stops decreasing and starts to go back up.
  - The model has started to overfit the training data
- In the early stopping, we stop training when the validation error reaches a minimum.

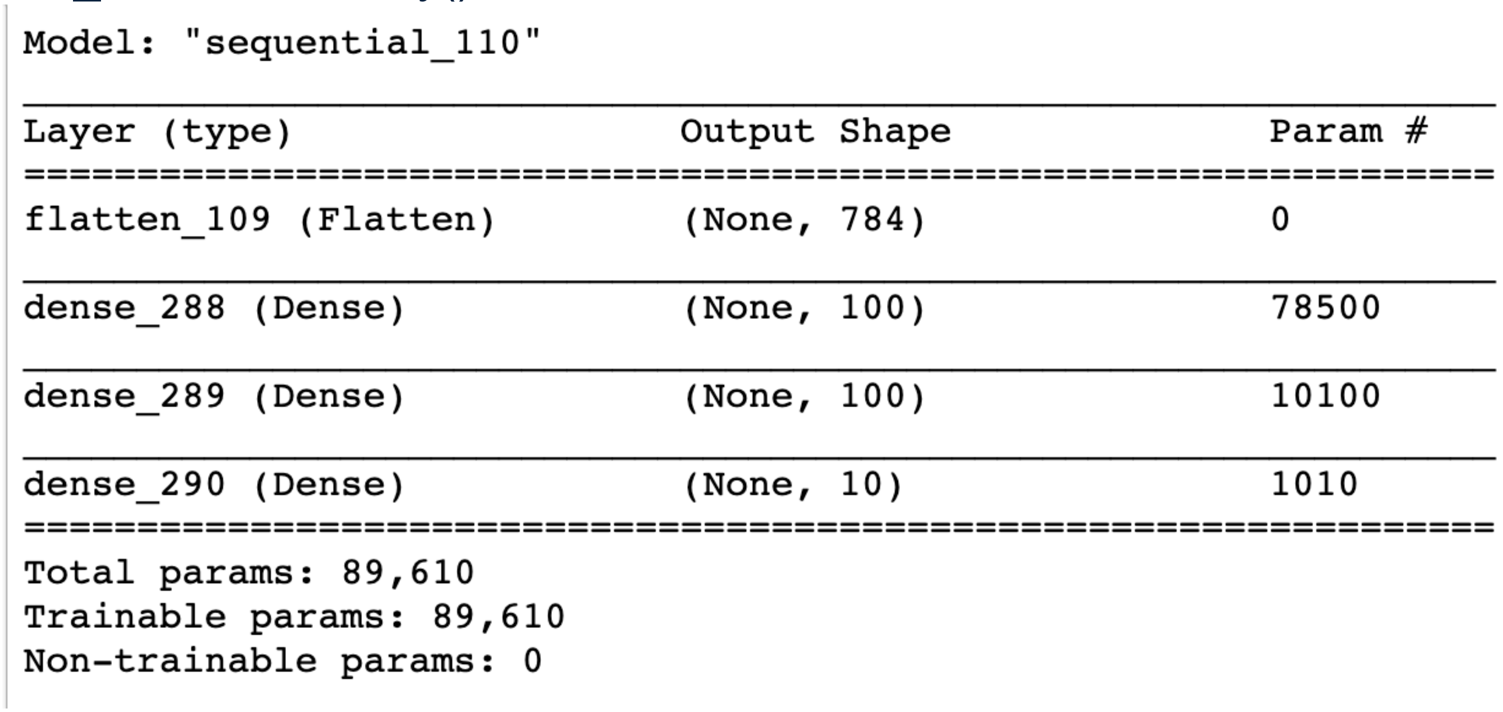


Fine-Tuning Neural Network Hyperparameters

- Get the best model (i.e., the best hyperparameter setting):

> best_model = rnd_search_cv.best_estimator_.model

> best_model.summary()



```
Model: "sequential_110"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
flatten_109 (Flatten)        (None, 784)               0         
_________________________________________________________________
dense_288 (Dense)            (None, 100)               78500     
_________________________________________________________________
dense_289 (Dense)            (None, 100)               10100     
_________________________________________________________________
dense_290 (Dense)            (None, 10)                1010      
=================================================================
Total params: 89,610
Trainable params: 89,610
Non-trainable params: 0
```

MLP for Regression in Keras

- Regression model can be built similarly. The main difference is the output layer (i.e., the activation function and loss function).

```
> X_reg_training,y_reg_training,X_reg_val,\
     X_reg_val,X_reg_test,y_reg_test = ... # as numpy arraws

> model_reg = keras.Sequential()
  # no need to use a Flattern layer for 1D data
  model_reg.add(keras.layers.Dense(4, activation="relu"))
  model_reg.add(keras.layers.Dense(1))
  # no need to specify metrics when compile if a linear activ. function is used
  model_reg.compile(optimizer="sgd", loss="mse")
  model_reg.fit(X_reg_training, y_reg_training,
          validation_data=(X_reg_val, y_reg_val), epochs=40)

> model_reg.evaluate(X_reg_test,y_reg_test)
  model_reg.predict(X_reg_test)
```

L1 and L2 Regularisation

- Deep neural networks often suffer overfitting.
  - "Deep" in terms of the hidden layer number
- One simple countermeasure is to penalise the large values of weights.
- Basic idea: Add an extra cost based on the (absolute) values in $\boldsymbol{W}$
- Flatten $\boldsymbol{W}$ as a 1-D array $(w_0, \ldots, w_n)$, let λ be a ***regularisation parameter***.
- L1 regularisation: $\tilde{J}(\boldsymbol{W}) = J(\boldsymbol{W}) + \lambda_1 \sum_{i=0}^{n} |w_i|$
  - where $\lambda_1 \sum_{i=0}^{n} |w_i|$ is the L1 regularisation term
- L2 regularisation: $\tilde{J}(\boldsymbol{W}) = J(\boldsymbol{W}) + \lambda_2 \sum_{i=0}^{n} {w_i}^2$
  - where $\lambda_2 \sum_{i=0}^{n} {w_i}^2$ is the L2 regularisation term
- Combination of L1 and L2: $\tilde{J}(\boldsymbol{W}) = J(\boldsymbol{W}) + \lambda_1 \sum_{i=0}^{n} |w_i| + \lambda_2 \sum_{i=0}^{n} {w_i}^2$

UNIVERSITY
OF WOLLONGONG
AUSTRALIA

L1, L2 and Minibatch Gradient Descent

```
> a, b = 0.01, 0.01
  n_records = 10

> model_reg = keras.Sequential()
  # no need to use a Flattern layer
  model_reg.add(keras.layers.Dense(4, activation="relu",
     kernel_regularizer=keras.regularizers.l1_l2(l1=a, l2=b)))
  model_reg.add(keras.layers.Dense(1))

> model_reg.compile(optimizer="sgd", loss="mse")
> model_reg.fit(X_reg_training, y_reg_training,
     validation_data=(X_reg_val, y_reg_val),
         epochs=40, batch_size=n_records)
```

Tools to Train Deep Neural Networks

Initialization Strategy

- Choose suitable initial weights and bias values in the network

```
> [name for name in dir(keras.initializers) if not name.startswith("_")]
  #['Constant', 'GlorotNormal', 'GlorotUniform','Identity', 'Initializer','Ones',...]
```

- Use initializer:

```
> keras.layers.Dense(10, activation="relu", kernel_initializer="he_normal")
```

❑ Official docs: https://keras.io/initializers/

Tools to Train Deep Neural Networks

Activation Functions

- We've introduced some activation functions such as Sigmoid, Tanh, ReLU, Softmax
- Get the list of activation functions in Keras

```
> [a for a in dir(keras.activations)]
  # [..., 'deserialize', 'elu', 'exponential', 'gelu', 'get', 'hard_sigmoid','linear',
  # 'relu', 'selu', 'serialize', 'sigmoid', 'softmax','softplus','softsign','swish',
  'tanh’]
```

❑ Official docs: https://keras.io/activations/

UNIVERSITY OF WOLLONGONG AUSTRALIA

Choose Faster Optimizers

Optimizers

- Standard gradient descent (incl. SGD and minibatch GD) works well in theory, but may not be efficient.
- There many *variants* of gradient descent, which can boost the convergence speed in training.
- To get the optimizer list:

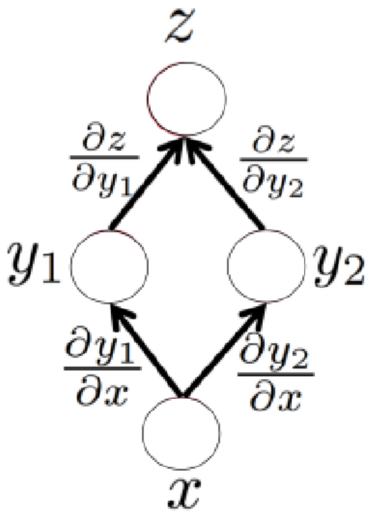
```
> [o for o in dir(keras.optimizers) ]
  # ['Adadelta','Adagrad', 'Adam', 'Adamax', 'Ftrl', 'Nadam', 'Optimizer',
  'RMSprop', 'SGD', ...]
```

- Beside choosing an optimizer, also need to determine hyperparameters in it.

❑ Official docs: https://keras.io/optimizers/

Appendix: Chain Rule of Calculus

- Two paths chain rule.
- $z = f(y_1, y_2)$ where $y_1 = g(x)$ and $y_2 = h(x)$
- $\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y_1}\frac{\partial y_1}{\partial x} + \frac{\partial z}{\partial y_2}\frac{\partial y_2}{\partial x}$ (Just identify ∂ as d in the previous slide.)



z
$\frac{\partial z}{\partial y_1}$
$\frac{\partial z}{\partial y_2}$
y_1
y_2
$\frac{\partial y_1}{\partial x}$
$\frac{\partial y_2}{\partial x}$
x

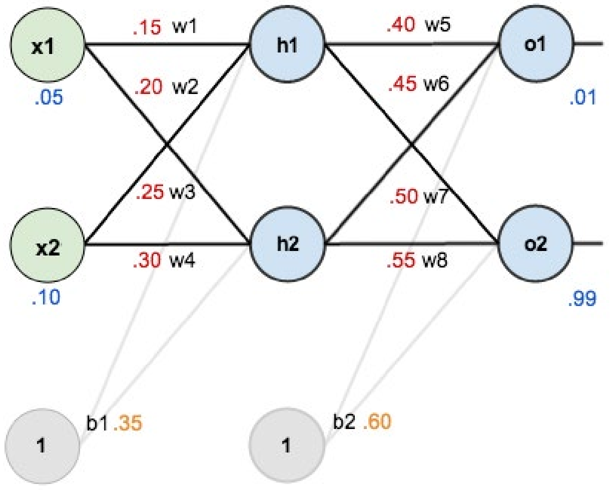
Appendix: How ANN is Trained

Backpropagation Training Algorithm.

- In general, backpropagation for MLPs comprises two steps: forward pass and backward pass

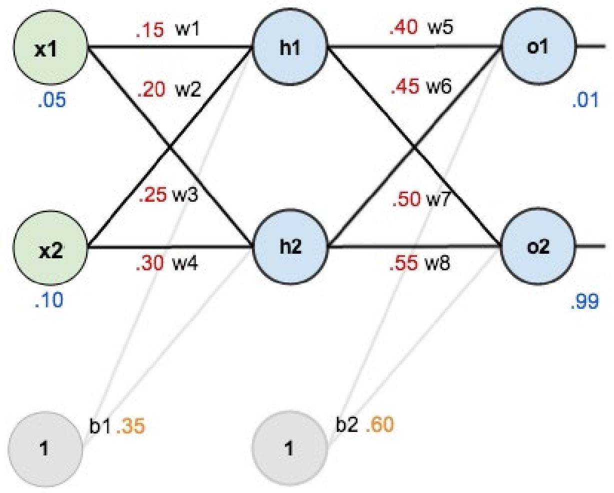
- **<u>Forward</u>**: calculates outputs given inputs.
  - For each training instance:

  1. *Feeds it to the network and computes the output of every neuron in each consecutive layer.*
  2. *Measures the network's output error (i.e., the difference between the true and the predicted output of the network)*
  3. *Computes how much each neuron in the last hidden layer contributed to each output neuron's error.*



Appendix: How ANN is Trained

- **Backward**:
  1. *Updates weights by calculating gradients.*
  2. *Measures how much of these error contributions came from each neuron in the previous hidden layer*
     - *Proceeds until the algorithm reaches the input layer.*
  3. *The last step is the gradient descent step on all the connection weights in the network, using the error gradients measure earlier.*



UNIVERSITY OF WOLLONGONG AUSTRALIA